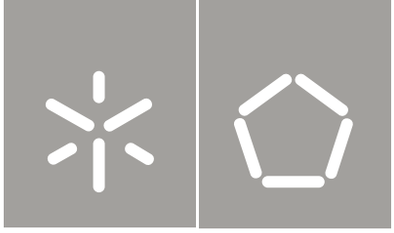
Universidade do Minho
Escola de Engenharia

Jorge Miguel Silva Aires

Migração de funcionalidades de um sistema operativo de tempo real para Gateware de um processador

Migração de funcionalidades de um sistema operativo de tempo real para Gateware de um processador
Jorge Miguel Silva Aires
UMinho | 2012

Outubro de 2012



Universidade do Minho
Escola de Engenharia

Jorge Miguel Silva Aires

Migração de funcionalidades de um sistema operativo de tempo real para Gateware de um processador

Tese de Mestrado
Ciclo de Estudos Integrados Conducentes ao Grau de
Mestre em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Paulo Cardoso

Outubro de 2012

Agradecimentos

Ao longo deste projeto, várias pessoas colaboraram tornando possível o seu desenvolvimento, às quais quero agradecer.

Em primeiro lugar gostaria de agradecer ao Professor doutor Adriano Tavares, pelo apoio prestado no desenvolvimento desta dissertação, assim como pelo conhecimento e os valores que adquiri ao frequentar as unidades curriculares, lecionadas pelo mesmo, no meu quarto ano de mestrado integrado

Gostaria de agradecer o meu orientador, o Professor Doutor Paulo Cardoso pela paciência e a preocupação diária que este teve em meu favor no decorrer deste projeto.

Queria agradecer também aos restantes professores do ESRG (*Embedded System Research Group*) com especial foco ao Professor Doutor Jorge Cabral que ocasionalmente me perguntava, com humor: "Está tudo bem Areias? Ah, desculpa Aires!"

Tenho que agradecer os maiores incentivadores desta epopeia, meus pais, José Aires e Adelina Silva, que apesar do longo período de ausência física que tivemos nunca deixaram de estar comigo a nível emocional. Também agradeço aos meus irmãos mais velhos, Filipe e Paula, que apesar de não perceber o meu trabalho, sempre se mostraram interessados e sempre me motivaram.

Por fim, gostaria de agradecer ao resto da comunidade ESRGiana (é deste modo que nós membros do grupo nos denominamos), alunos de licenciatura, mestrado e doutoramento, pela ajuda, pelas brincadeiras e pelo convívio realizado no dia-a-dia, que fizeram com que estes anos passassem sem serem vistos.

Jorge Aires

Resumo

Migração de funcionalidades de um sistema operativo de tempo real para Gateware de um processador.

Palavras-chaves: *Real Time Operating Systems* (RTOS); Desenho de processadores; FPGA; Gateware *migration*.

Nesta dissertação pretende-se efetuar a migração de uma ou diversas funcionalidades de um sistema operativo de tempo real, para *hardware* de um microprocessador, sob a forma de unidade funcional, e analisar a viabilidade deste tipo de abordagem, criando assim um sistema embebido *real-time* reconfigurável e determinístico.

Neste sentido é realizado o estudo aprofundado de um sistema operativo tempo real de modo a identificar e extrair as suas funcionalidades para posteriormente serem implementadas em *hardware*. O escalonador de tarefas e os mecanismos de sincronismo (*mutex/semaphore*) são exemplos de funcionalidades a implementar. O processador que será usado para receber as novas funcionalidades deverá possuir uma arquitetura RISC com 32 bits de tamanho de instruções, para uma implementação mais acessível e abrangente. Este será estudado o suficiente para entender o seu modo de execução, assim como, determinar os módulos que possui e como estes estão interligados.

Tendo em conta a relação entre as três métricas escolhidas (desempenho temporal, unidades lógicas usadas e consumo de energia) para o desenvolvimento do projeto, este trabalho demonstra a viabilidade da passagem de uma ou diversas dessas funcionalidades para *hardware* a fim de tornar o sistema embebido dedicado a certas aplicações.

Abstract

Real-Time Operating System - Gateware Migration Environment

Keywords: Real Time Operating Systems (RTOS); Processors Design; FPGA; Gateware migration.

This thesis aims to migrate one or several features, of a real time operating system, to microprocessor gateware and analyze the feasibility of this approach by creating a reconfigurable and deterministic real-time embedded system.

Towards this goal, in-depth study of a real-time operating system will be performed in order to identify and extract its features, to be implemented later in hardware. The task scheduler and the synchronization mechanisms (mutex / semaphore) are examples of features to implement. The processor used to integrate the new features should have a 32-bit RISC architecture. This should be studied enough to understand its mode of execution, as well as determine which modules it has and how they are interlinked.

Taking into account the relationship between the three chosen metrics (performance, logic unit used and energy consumption) for the development of the project, proving the viability of the migration of one or several features to the processor gateware in order to make the embedded system dedicated to some applications will be attempted.

Índice

Agradecimentos iii

Resumo v

Abstract vii

Capítulo 1 Introdução 1

1.1 Enquadramento 1

1.2 Objetivos 2

1.3 Organização da Dissertação 3

Capítulo 2 Caracterização do Projeto 5

2.1 Condições do projeto 5

2.1.1 Restrições do projeto 5

2.1.2 Requisitos do projeto 6

2.2 Seleção do ambiente de execução 7

2.2.1 Análise e comparação dos RTOS 7

2.2.2 Análise e comparação dos processadores 11

2.2.3 Discussão 12

2.3 Exemplos e Técnicas de Migrações de *Software* para *Hardware* 13

2.3.1 Escalonador Pfair 13

2.3.2 RTBlaze 14

2.3.3 O ARPA-MT 16

2.3.4 ARTESSO *hardware* RTOS 18

2.3.5 Conclusão 19

Capítulo 3 Tecnologias de suporte à migração 21

3.1 O sistema operativo de tempo real – eCos™ 21
3.1.1 *Hardware Abstraction Layer* (HAL) 23
3.1.2 O *kernel* 28
3.1.3 Gestão das Exceções e Interrupções 38
3.2 O Processador RISC – OR1200 42
3.2.1 Unidade Central de Processamento - CPU 43
3.2.2 *Memory Managment Unit* e caches 44
3.2.3 O barramento WISHBONE e sua interface 46
3.2.4 Os restantes periféricos opcionais 49
3.3 ORPSoC – *OpenRisc reference Platform System-on-Chip* 52
3.3.1 Os *arbiters* do barramento de dados e de instruções 52
3.3.2 Memória Externa 54
3.3.3 O módulo UART 16550 55
Capitulo 4 Modelação e Implementação 57
4.1 Migração de funções do RTOS para *hardware* 57
4.1.1 Pré-requisitos de *Software* 58
4.1.2 O módulo *hardware* RTOS – Hrtos 60
4.1.3 Funções a migrar para *hardware* 68
4.2 Implementação 75
Capítulo 5 Resultados Experimentais 81
5.1 Metodologias de Teste 81
5.2 Testes e Resultados 83
5.2.1 Teste de funcionamento 83
5.2.2 Testes de desempenho temporal 86

5.2.3 Resultados das unidades lógicas usadas 87
5.2.4 Resultados da energia dissipada 89
5.3 Análise dos resultados 90
Capítulo 6 Conclusões 93
6.1 Conclusões 93
6.2 Trabalho Futuro 94
Referências 97
Bibliografia 99
ANEXOS 101

Lista de Abreviaturas e Siglas

ABEL - Advanced Boolean Expression Language

AMBA – Advanced Microcontroller Bus Architecture

API – Application Programming Interface

ASIC – Application Specific Integrated Circuit

CCU – Custom Compute Unit

CLBs – Configurable Logic Blocks

CPU – Central Processing Unit

DBus – Data Bus

DMA – Direct Memory Access

DSR – Deferred Service Routine

ECOS- Embedded Configurable Operating System

E/S – Entradas e saídas

FIFO – First In First Out

FPGA – Field Programmable Gate Array

FPU – Floating Point Unit

FSM – Finite State Machine

GDB – GNU debugger

GPS - Global Positioning System

GUI – Graphical User Interface

HAL – Hardware Abstraction Layer

HDL – Hardware Description Languages

I/O – Input/ Output

I2C - Inter-Integrated Circuit

IDE – Integrated Development Environment

IP core – Intellectual Property core

ISA – Instruction Set Architecture

JTAG – Joint Test Access Group

LED - Light Emitting Diode

LUTs – Lookup Tables

MAC unit- Multiply and Accumulate unit

MIPS – Microprocessor without Interlocked Pipeline Stage

MMU – Memory Management Unit

PCP – Priority Ceiling Protocol

PIP – Priority Inheritance Protocol

RAM - Random Access Memory

RISC - Reduced Instruction Set Computer

RTOS – Real Time Operating System

RTU – Real-Time Unit

RTL- Register Transfer Level

SOC – System-on-chip

SPARC – Scalable Processor ARChitecture

SPI - Serial Peripheral Interface Bus

UART – Universal Asynchronous Receiver Transmitter

TLB – Translation Look aside Table

VHDL – Very high speed integrated circuit Hardware Description Languages

VSR – Vector Service Routine

Índice de Figuras

Figura 1: Placa Virtex 5 ML509 7

Figura 2: Escalonador Pfair 14

Figura 3: Processador Base com *tightly-coupled hardware* RTOS [4] 15

Figura 4: Visão geral do projeto ARPA-MT. CPU (MIPS32), Cop0-MEC (*memory managment unit*, configuração e *handling* interrupção e exceções), Cop2-OSC (*hardware* RTOS) [1]. 16

Figura 5: Diagrama de blocos internos do módulo Corp2-OSC [1] 17

Figura 6: Configuração do ARTESSO, e arquitetura do *hardware* RTOS [11] 18

Figura 7: Exemplificação das camadas constituintes do eCos™ [12] 22

Figura 8: Exemplificação da estrutura da diretoria do HAL do eCos™. 24

Figura 9: Constituição do HAL para a arquitetura OpenRISC 25

Figura 10: Etapas da inicialização do HAL [12] 27

Figura 11: Exemplo do escalonamento operado pelo escalonador *Multilevel queue*. 30

Figura 12: Diagrama de classes - escalonador *Bitmap* 31

Figura 13: Diagrama de classes - *threads* 32

Figura 14: Diagrama de classes – lista de *threads* 33

Figura 15: Gestão das *Threads* 34

Figura 16: Efeito de *priority inversion* (A) e *priority inheritance protocol* (B) 35

Figura 17: Inicialização do *kernel* do eCos™ 37

Figura 18: Gestão de uma exceção pelo eCos™ [12] 39

Figura 19: Manipulação de uma interrupção no eCos™ [12]. 41

Figura 20: Diagrama de blocos - constituição do Processador OR1200 [13] 42

Figura 21: Diagrama de Blocos - constituição interna do CPU do OR1200. 43

Figura 22: Diagrama de blocos MMU – Tradução do endereço efetivo para o endereço físico [14] 45

Figura 23- Esquema de ligação a barramento Wishbone [15] 47

Figura 24*:* Wishbone *Master Signal*- ciclo de leitura e escrita 48

Figura 25: Diagrama de Blocos- Temporizador/Contador [14] 49

Figura 26: Diagrama de blocos- Unidade de *Debug* [14] 50

Figura 27: Diagrama de blocos- *Programmable Interrupt Controller (PIC)* [14] 51

Figura 28: Diagrama de blocos - Ligações dos *arbiters* entre *slave* e *master* 53

Figura 29: Diagrama de blocos- memória externa de dados e de instruções 54

Figura 30: Diagrama de blocos- módulo UART do OR1200 [17] 55

Figura 31: Exemplificação da alteração das bibliotecas do eCos™ 60

Figura 32: Funcionamento do Hrtos 61

Figura 33: Esquema da unidade Hrtos_Decode 62

Figura 34: Diagrama de estados do sinal 'insn_rtos_reg' 63

Figura 35: Esquema de ligação entre *Register File* e Hrtos 64

Figura 36: Esquema de ligação entre Hrtos e a memória de dados 66

Figura 37: Esquema de ligação entre Hrtos e os registos especiais 67

Figura 38: Código da função 'hal_thread_load_context' em *software*. 69

Figura 39: Instrução para ativar a função 'load_context' em *hardware* 70

Figura 40: Diagrama de estados da função 'load_context' em *hardware*. 71

Figura 41: Código da função 'hal_thread_context_switch' em *software*. 72

Figura 42: Diagrama de estados da função 'context_switch' em *hardware*. 73

Figura 43: Instrução para ativar 'context_switch' em *hardware* 74

Figura 44: Diagrama de estado para troca entre funções 74

Figura 45: RTL – Hrtos 77

Figura 46: RTL –HTHREAD 78
Figura 47: RTL - CPU e Hrtos 79
Figura 48: Código principal usado para efetuar as simulações 82
Figura 49: Execução da função 'context_switch' pelo Hrtos 83
Figura 50: Ocorrência de interrupção quando Hrtos está em execução 85
Figura 51: Tempo de execução das funções 'load_context' e 'context_switch' com e sem Hrtos .. 86
Figura 52: Tempo de finalização do programa usando o escalonador Bitmap e Mlqueue. 86
Figura 53: Unidades lógicas totais utilizada por OR1200 com e sem Hrtos 87
Figura 54: Ocupação percentual de OR1200 com e sem Hrtos na placa 87
Figura 55: Distribuição por módulos das unidades lógicas utilizadas por Hrtos 88
Figura 56: Distribuição percentual das unidades lógicas utilizadas pelos módulos de Hrtos 88
Figura 57: Potência consumida por OR1200 com e sem Hrtos 89
Figura 58: Energia dissipada total por OR1200 com e sem Hrtos, ... 89
Figura 59: GUI Configtool – Janela principal 104
Figura 60: GUI Configtool- localizar ferramentas 105
Figura 61: GUI Configtool- localizar repositório 105
Figura 62: GUI Configtool- Janela de resolução de conflitos 106
Figura 63: Código da ferramenta 'vmem2coe' 111
Figura 64: Localização da ferramenta I/O Pin Planning 112
Figura 65: Ferramenta 'PlanAhead' 112
Figura 66: Localização da ferramenta 'iMPACT'. 113
Figura 67: Configuração dos pinos SW3 da placa XC5VLX110T e ligação JTAG. 113
Figura 68: iMPACT- Inicializar 114
Figura 69: iMPACT- Selecionar ficheiro do 'Design' 114

Figura 70: iMPACT – Programação da placa 115

Figura 71: Esquema de ligação a placa XC5VLX110T 115

Índice de Tabelas

Tabela 1: RTOS e respetivos *porting*, para diversos processadores 8

Tabela 2: Comparação dos RTOS sobre as suas capacidades 10

Tabela 3: Comparação entre o Leon2 e o OR1200 12

Tabela 4: Tabela VSR do eCos™ para o OpenRISC 38

Tabela 5: Instruções adicionadas ao ISA OR1200 59

Tabela 6: Registos de Prepósito Geral (GPR) do OR1200 65

Tabela 7: Output da função 'Load_context' e 'switch_context' 84

Capítulo 1

Introdução

Neste capítulo é apresentado o enquadramento desta dissertação, assim como os seus objetivos e contribuição no domínio científico. Para finalizar será identificada a organização da dissertação.

1.1 Enquadramento

Atualmente os sistemas embebidos são omnipresentes, desde o *pacemaker* até ao sistema de controlo de um satélite, passando pelo telemóvel até ao leitor mp3; estes dispositivos são utilizados diariamente pela população em diversas áreas de consumo. Alguns destes sistemas são capazes de executar múltiplas tarefas; para tal, muitos deles dispõem no seu *software* de um sistema operativo de tempo real (RTOS).

Os RTOS são destinados à execução de múltiplas tarefas em que o seu tempo de execução tem que ser cumprido dentro do prazo máximo esperado, não obstando o seu comportamento funcional e temporal. Os RTOSs devem ser tão determinísticos quanto imposto pelas características do sistema. No entanto, a utilização de um RTOS em *software*, num sistema embebido que deve ser hard real-time, encontra dois problemas críticos: o acréscimo de latência na ocorrência de interrupção e o *overhead* da operação de *context switch*. O primeiro problema deve-se ao tempo que o RTOS leva a executar o serviço a uma interrupção quando esta já foi ativada, enquanto o

segundo problema, traduzido para instruções a realizar, consome muito tempo útil do processador. Estes problemas somados penalizam o desempenho do sistema embebido. De forma a minimizar estes efeitos desagradáveis, projetistas têm investigado a migração de vários componentes básicos de um RTOS para *hardware* [1] [2] [3] [4].

Estes componentes de *hardware*, designados de *Intellectual Property* (IP) *cores* podem ser acoplados ao processador de duas formas: *loosely-coupled* e *tightly-coupled*. A primeira abordagem permite a utilização de qualquer CPU, visto que o IP *core* está ligado ao barramento externo do CPU, porém não garante que o sistema final seja completamente determinístico. A outra abordagem, que consiste em acoplar o IP *core* ao processador com um barramento especial, permite tornar o sistema mais determinístico, no entanto, torna a sua migração para outros processadores mais complexa e consequentemente mais cara.

Com a aparição das *Field Programmable Gate Array* (FPGA) e das linguagens de descrição de *hardware* (HDL), o desenvolvimento destes IP *core* tornou-se muito mais fácil e rápido, bem como a criação de processadores customizados.

Com o objetivo de contribuir para a evolução das técnicas de desenho de processadores, a abordagem escolhida consistirá em efetuar a migração de funções e estruturas de um sistema operativo de tempo real, tais como o escalonador de tarefas, *semaphores* ou *threads,* para gateware de um processador e verificar a sua viabilidade, tendo em conta as métricas desempenho temporal, unidades lógicas utilizadas e energia dissipada.

1.2 Objetivos

O objetivo desta dissertação consiste em efetuar a migração de algumas funcionalidades e estruturas de um sistema operativo de tempo real, existente no mercado, para o *gateware* de um microprocessador, criando um sistema embebido *real-time* reconfigurável e determinístico, capaz de se adequar a várias aplicações.

Numa primeira fase será estudada a constituição do processador assim como a do RTOS selecionado, de seguida serão identificadas e implementadas algumas funcionalidades do RTOS

em *hardware,* que serão acopladas ao processador. Estas são codificadas usando uma linguagem de descrição de *hardware* e sintetizadas com recurso a uma FPGA.

Numa segunda fase, será analisada a viabilidade deste tipo de abordagem tendo em conta as três métricas de modelação: performance temporal, dissipação de energia e unidades lógicas utilizadas.

1.3 Organização da Dissertação

Os temas abordados nesta dissertação estão divididos em seis capítulos e organizados tal como de seguida apresentado.

No segundo capítulo é inicialmente efetuada uma análise do problema, em relação aos objetivos propostos. Depois são apresentadas as restrições do projeto seguido dos requisitos que este deve possuir. Seguidamente é discutido qual será o sistema operativo de tempo real a utilizar, assim como o respetivo processador. Na parte final é feita uma apresentação de diversos exemplos e técnicas de migração de *software* para *hardware* efetuadas até ao momento na área específica.

No terceiro capítulo são apresentadas as tecnologias utilizadas para efetuar as migrações, isto é, o RTOS escolhido e o respetivo processador assim como os restantes periféricos que em conjunto formaram o sistema embebido base.

No quarto capítulo são apresentadas as especificações das migrações de funcionalidades, assim como esta se encaixam no funcionamento do processador e respetivo RTOS híbrido. No final deste capitulo também é apresentada a implementação final do sistema recorrendo a imagens.

No quinto capítulo é apresentado a metodologia dos testes efetuados, seguidas dos resultados a nível de funcionamento, performance temporal, unidades lógicas utilizadas e consumo de energia. Para finalizar é efetuada a análise dos resultados obtidos.

O sexto e último capítulo apresenta as conclusões do projeto assim como as perspetivas futuras do trabalho descrito nesta dissertação.

Capítulo 2

Caracterização do Projeto

Neste Capitulo, tendo em conta os objetivos propostos, serão abordados as restrições do projeto, assim como os requisitos que este terá de cumprir. Também será esclarecido, tendo em conta essas duas premissas, a escolha do RTOS e do respetivo processador para a realização do projeto. Para finalizar serão apresentados algumas técnicas e exemplos de migração, de *software* para *hardware* que servirão de base para caracterizar o trabalho desta dissertação.

2.1 Condições do projeto

Quando se está a planear um projeto, após saber os objetivos, a identificação das restrições e os requisitos/condições impostos por um cliente permitem realizar um guia simples para uma boa implementação. Assim neste subcapítulo serão apresentados as condições necessárias para a realização deste projeto, assim como as limitações/restrições iniciais ao desenvolvimento deste.

2.1.1 Restrições do projeto

Tendo em conta os objetivos e a disponibilidade de recursos, o projeto a desenvolver terá as seguintes restrições:

- O processador que irá ser usado, para implementar as novas funcionalidades, deverá ser de licença aberta, possuir uma versão numa linguagem de descrição de *hardware* (HDL), o Verilog, e possuir um formato de instruções com tamanho de 32bit;
- O sistema operativo que irá ser usado deverá ser *real-time*. Em relação à disponibilidade do código fonte este terá de ser de licença aberta. O RTOS deverá possuir ferramentas de suporte e desenvolvimento para o processador escolhido, isto é, deverá existir um *port* do RTOS para o processador que deverá incluir ferramentas como compiladores C, assemblador e *linker,* que construirão as bibliotecas adequadas para o processador escolhido. O número de tarefas suportadas pelo RTOS deverá ser superior a duas, para efeitos de *multithreading*.

2.1.2 Requisitos do projeto

A nível de requisitos do projeto, estes dividem-se em dois grupos: os requisitos funcionais e os requisitos não funcionais.

Os requisitos funcionais são os requisitos relacionados com as ações que um sistema deverá ser capaz de executar, assim como o comportamento das entradas e saídas que este apresentará. Assim ao nível da aplicação, as funcionalidades migradas para *hardware* deverão possuir os mesmos comportamentos do que a funcionalidade original do RTOS. Por exemplo, se a função tinha que ativar um LED, está terá que continuar a fazê-lo independentemente de estar implementada em *hardware* ou em *software*. O sistema embebido final deverá possuir uma ferramenta de configuração, onde será possível escolher que funcionalidades migrar (nenhuma ou diversas), pois sendo um dos objetivos tornar o sistema embebido reconfigurável, é necessário que possua esta ferramenta de modelação. A nível de interface com utilizador, dependendo do programa, o sistema terá que ter botões, *switch* e LED mapeados, para posteriormente efetuar a verificação e testes de funcionamento do mesmo. Também poderá ser acrescentado um módulo de interface série.

Os requisitos não funcionais são requisitos que o projeto deve possuir, mas que não estão relacionados com a sua utilidade prática. Sendo assim, um dos requisitos deste projeto é encontrar o *trade-off* entre as métricas performance, energia consumida e Configurable Logic Block (CLBs)

usadas. Outro requisito que o sistema deverá cumprir é que este terá que ser determinístico, isto é, um sistema que produzirá sempre o mesmo comportamento na saída, para uma determinada entrada. Por fim o ultimo requisito que este deverá satisfazer, é que toda a sua implementação deverá ser capaz de ser sintetizada e testada na plataforma XC5VLX110T [5] da Xilinx, exibida na Figura 1.

Figura 1: Placa Virtex 5 ML509

2.2 Seleção do ambiente de execução

Neste subcapítulo apresenta-se o estudo que se realizou para determinar qual seria o RTOS a escolher assim como o respetivo processador, com base nas restrições e nos requisitos acima referidos.

2.2.1 Análise e comparação dos RTOS

Existem múltiplos RTOS de licença livre no mercado com variadas especificações e implementações. Mais de 25 RTOS de licença aberta podem ser obtidos gratuitamente na internet, mas foram escolhidos apenas três para analisar, de modo a encontrar qual será o alvo de estudo para o desenvolvimento do projeto: O FreeRTOS™, o eCos™ e o ChibiOS/RT.

O FreeRTOS™ [6] é um RTOS desenvolvido por um grupo denominado de *Real Time Engineers Ltd*. Este RTOS é largamente utilizado em diversos sistemas embebidos, assim como em diversos projetos de investigação. O código fonte deste RTOS está desenvolvido em C. Atualmente este RTOS possui 31 *ports* para arquiteturas diferentes, desde processadores de 8-bit a 64-bit, ver Tabela 1. Este elevado número de *ports* é sem dúvida uma das suas grandes vantagens, no entanto para usufruir do manual do utilizador, este terá que ser pago num valor de 30$, o que pode ser um entrave ao uso deste RTOS.

Tabela 1: RTOS e respetivos *porting*, para diversos processadores

Arquitetura/Variante	FreeRTOS™	ChibiOS/RT	eCos™
SPARC LEON 2			X
x86			X
AVR32	X		
Cortex-M3	X	X	
MIPS 32			X
OR1200			X
STM32	X	X	
PowerPC		X	X
IA32	X		X
MSP430	X	X	
ColdFire	X	X	X
H8S	X	X	X
PIC 32	X		
μBlase	X		
OUTROS...	X	X	X

O eCos™ (*embedded Configurable operating system*) [7] desenvolvido pela *Cygnus Solutions* e mantido atualmente pela empresa *RedHat*, é um sistema operativo real-time que possui a característica de ter um *kernel* modulável, isto é, possui uma ferramenta que lhe permite optar por certas bibliotecas (entre outras a biblioteca POSIX) e funcionalidades em detrimento de outros, através do mesmo código fonte. Estas opções são selecionadas de uma forma simples e eficaz recorrendo a uma *Graphical User Interface* (GUI) denominada *Configtool*. As ferramentas de configuração assim como o código fonte estão desenvolvidas em C/C++ e permitem serem compiladas tanto em Windows[1] como em Linux. A documentação fornecida para este RTOS está bastante desenvolvida, mas carece em alguns aspetos tais como no *User Guide* que não está totalmente atualizado. No entanto, este RTOS é sem dúvida um ótimo candidato para implementação do projeto, por possuir diversos *ports* para diversos processadores de código aberto e por ser altamente configurável.

O ChibiOS/RT [8], desenvolvido pela própria comunidade 'chibios.org', trata-se de um sistema operativo real-time compacto (quando todas as bibliotecas são compiladas, o seu tamanho em memória não excede os 6kBytes) e simplista. Este RTOS desenvolvido em C, só possui *ports* para 8 arquiteturas diferentes o que em comparação com o FreeRTOS™, resume-se a poucas opções. A documentação disponibilizada está fragmentada por cada *port* existente, sendo que para cada um existe um manual de referência para o *kernel* e outro para o respetivo HAL (*Hardware Abstraction Layer*). Este RTOS apesar de ser bastante compacto, não apresenta nenhuma vantagem direta sobre os restantes apresentados.

Após efetuar uma breve descrição dos três RTOS selecionados, referindo o número de *ports*, a linguagem do seu código fonte e alguns aspetos não funcionais, realizou-se uma comparação destes RTOS em relação as capacidades do *kernel*, do escalonador, das *threads* e da gestão da memória virtual, representada na Tabela 2.

[1] Com auxílio a ferramenta CygWin.

Tabela 2: Comparação dos RTOS sobre as suas capacidades

	FreeRTOS™	ChibiOS/RT	eCos™
		KERNEL	
Multithreading	SIM	SIM	SIM
Estático	SIM	SIM	SIM
		ESCALONADOR	
Tipos	Round-Robin	Round-Robin	Bitmap/Mlqueue
Preemptivo	SIM	SIM	SIM
		THREADS	
Nível Prioridades	0-32	0-128	0-32
Nº Max Threads	+32	64	+ 32
Troca prioridades	SIM	SIM	SIM
		MEMÓRIA	
Alocação Dinâmica	SIM	-	SIM
Gestão Memória Virtual	SIM	SIM	SIM
Mutex/Semaphore	SIM/SIM	SIM/SIM	SIM/SIM

Com base nesta comparação verifica-se que todos os RTOS têm a capacidade de efetuar *multithreading*, isto é, conseguem executar diversas tarefas em simultâneo. Também se pode afirmar que, como possuem um *kernel* estático, todos eles após efetuar a compilação do seu código fonte não podem adicionar novos módulos, sem efetuar a compilação novamente, isto permite que o *kernel* destes RTOS seja mais especifico a arquitetura da plataforma/processador escolhido. A nível de escalonadores pode-se constatar que todos eles fornecem escalonadores preemptivos o que lhes permite parar a execução de uma tarefa, para executar outra com uma prioridade superior que esteja pronta para ser executada, efetuando assim uma comutação de contexto. É de salientar que todos têm um tipo de escalonador com *timeslice*, este permite-lhes dar uma porção de tempo de execução a cada tarefa, no entanto esta opção pode não ser vantajosa, pois se o tempo dado for inferior ao tempo necessário para efetuar a comutação de contexto, há grandes probabilidades de sobrecarregar o CPU e de afetar o seu desempenho. Cada RTOS apresentado consegue gerir mais de 32 *threads,* sendo que cada um deles tem a possibilidade de

variar a prioridade dessas de 0 até 32. Mais concretamente o chibiOS/RT consegue ir até ao nível de prioridade 128 enquanto o FreeRTOS™ permite ao utilizador inserir o numero de *threads* que quiser, assim como a escala de níveis de prioridade que o utilizador desejar. É de salientar que o eCos™ e o FreeRTOS™ permitem *threads* com o mesmo nível de prioridade, enquanto o chibiOS/RT não. A nível de gestão da memória, os RTOS permitem alocar a memória virtualmente e de uma forma dinâmica. Também possuem mecanismos de sincronização entre tarefas, como *mutexes* e *semaphores*, para efetuar acessos protegidos a memória ou a determinadas funções.

Efetuando uma análise crítica do que foi analisado, todos os três RTOS apresentados são bons candidatos para a implementação deste projeto, no entanto só com a análise efetuada no próximo subcapítulo é que se poderá chegar a uma conclusão, sobre qual, entre o FreeRTOS™, o ChibiOS/RT e o eCos™, se deve escolher.

2.2.2 Análise e comparação dos processadores

Com base nos RTOS escolhido é realizado agora a escolha do processador para qual já existe *port*, que servirá de base para o resto do projeto. Com base na Tabela 1, verifica-se, que apesar de existirem diversos processadores para qual existe o *port* dos três RTOS mencionados, muitos deste não possuem uma versão gratuita e completa do código em linguagem de descrição de *hardware*. Sendo assim, o número de escolhas do processador está reduzido a dois: o Sparc LEON 2 e o OpenRisc 1200.

O Sparc Leon 2 foi desenvolvido pelo grupo *Gaisler Research* [9]. Este processador possui uma arquitetura RISC com 32-bit de tamanho de instrução. O CPU possui 5 estágios no seu *pipeline*, esta ainda possui uma cache de instrução e de dados separados e *Memory Managment Unit*. Este processador vem com diversos periféricos embutidos, tratando-se assim de um *System-on-chip.* Vem com porta série, portos I/O, timers e controlador de interrupção. A conexão entre o CPU e os periféricos é efetuada usando o barramento AMBA. A linguagem de descrição de *hardware* para a qual já foi implementado este SOC é o *Very high speed integrated circuit Hardware Description Languages* (VHDL).

O OpenRISC1200 é um processador desenvolvido pela comunidade *OpenCores*. Este processador possui as mesmas características mencionadas para o Leon 2, com exceção de que este

processador possui um barramento *Wishbone*, para conexão com seus periféricos externos. Este processador foi desenvolvido com uma linguagem HDL diferente do que o LEON2, o Verilog,

2.2.3 Discussão

Com base nas restrições mencionadas na secção 2.1.1 e 2.1.2 e na análise efetuada nos dois subcapítulos anteriores verifica-se os resultados apresentados na Tabela 3.

Tabela 3: Comparação entre o Leon2 e o OR1200

	Open Source	Tamanho Instrução	Arquitetura RISC	RTOS port	Linguagem HDL
SPARC LEON	Sim	32-bit	Sim	eCos™	VHDL
OR1200	Sim	32-bit	Sim	eCos™	Verilog HDL

Com base na Tabela 1 e na análise efetuada em 2.2.1, pode-se afirmar que o RTOS a utilizar será o eCos™, visto a ser o único que possui *port*, até ao momento, para estes dois processadores.

Em relação aos outros parâmetros que caracterizam os processadores, verifica-se que ambos possuem uma arquitetura RISC e ambos possuem um formato de instrução de 32-bit. O único parâmetro que os pode diferenciar é o tipo de linguagem HDL, no qual existe uma versão do código do processador para FPGA. Visto que o Verilog é uma linguagem HDL mais simples e sendo um dos requisitos do projeto o processador escolhido é o OpenRISC 1200, pois com base na Tabela 3, é o único processador que está disponível na linguagem de descrição de *hardware* – Verilog.

Concluindo a Análise dos processadores e do RTOS respetivo, ficamos então a saber que para a realização da parte prática desta dissertação serão usados o processador OpenRISC 1200 e o RTOS eCos™.

2.3 Exemplos e Técnicas de Migrações de *Software* para *Hardware*

Existem diversas migrações de funções/métodos em *software* para *hardware* com finalidade de aumentar a performance de um sistema embebido. Este conceito não é novo [3] [10], mas só nos nossos dias, a tecnologia e os meios permitem apostar com abundância neste conceito de implementação.

No desenvolvimento de novos módulos/CCUs ou coprocessadores, estes podem ser *loosely-coupled* [1] [2] [3]ou *tightly-coupled* [4] ao processador principal. O termo *tightly coupled* refere-se às unidades de *hardware*, geralmente CCUs, que estão intrinsecamente ligadas ao processador central, partilhando barramentos e/ou a memória interna, sendo que a sua estrutura tem uma forte dependência da estrutura do processador, dificultando a sua reutilização para outras arquiteturas de processadores. O termo *loosely-coupled* refere-se às unidades, que apesar de comunicarem uma com as outras e/ou com o processador central através de um barramento especial (AMBA, Wishbone, etc), são totalmente independentes a nível funcional e estrutural, permitindo que sejam exportadas para outras arquiteturas de uma forma mais simples e mais rápida.

Tendo estes dois conceitos de implementação em mente, vai-se agora analisar diversos exemplos de migrações e implementações em *hardware* efetuadas até ao momento.

2.3.1 Escalonador Pfair

O Pfair [2] é um escalonador em *hardware* para multiprocessadores de tempo real implementado por Nikhil Gupta. Funciona como um coprocessador, *loosely-coupled*, que executa o algoritmo de escalonamento e determina qual será a próxima tarefa a ser executa para cada um dos núcleos do multiprocessador.

O escalonador Pfair está dividido em quatro blocos: O registo de estados das tarefas (TR), a calculadora de ordem parcial (POC), o gerador de escalonamento (SG) e o controlador geral (MC), apresentados na Figura 2. O Registo de estados das tarefas permite guardar a prioridade da tarefa, este pormenor é importante porque permite ao escalonador ser preemptivo. A calculadora de

ordem parcial e o gerador de escalonamento, permitem calcular a nova *thread* a ser executada, sem atulhar o processador de instruções; esta funcionalidade diminui o *overhead* na ocorrência de *context switch*. O último bloco é responsável por controlar todos os sinais de entrada e saída para o Pfair.

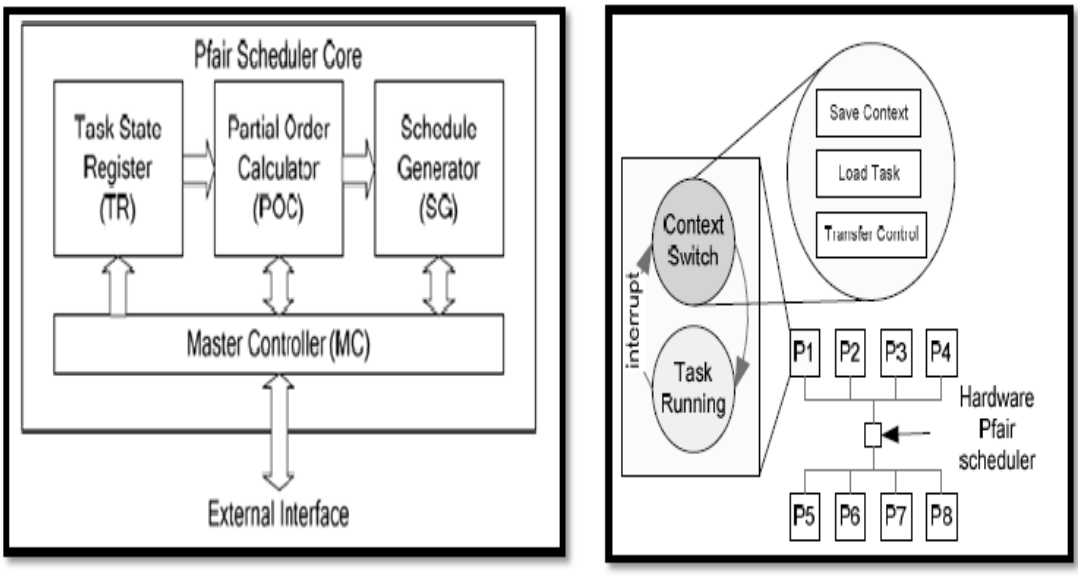


Figura 2: Escalonador Pfair diagrama de blocos (esquerda), funcionamento do Pfair (direita) [2]

Segundo o autor, esta implementação de escalonador permite aumentar a performance do sistema embebido, em relação ao tempo perdido no *context switch*, assim como a energia dissipada em comparação com escalonadores em *software* e escalonadores a correr num único núcleo aquando presentes numa implementação *multi-processor*.

2.3.2 RTBlaze

O RTBlaze [4] trata-se de um projeto onde foi desenvolvido um processador com um *tightly-coupled hardware* RTOS com interface dedicada. Este projeto desenvolvido por Terance Wijesenghe possui um escalonador preemptivo, *threads, timer* e *semaphore* implementados em *hardware*. A Figura 3 ilustra como se encaixa o módulos do *hardware* RTOS no *pipeline* do processador (a verde, os módulos das funcionalidades do RTOS migradas para *hardware*).

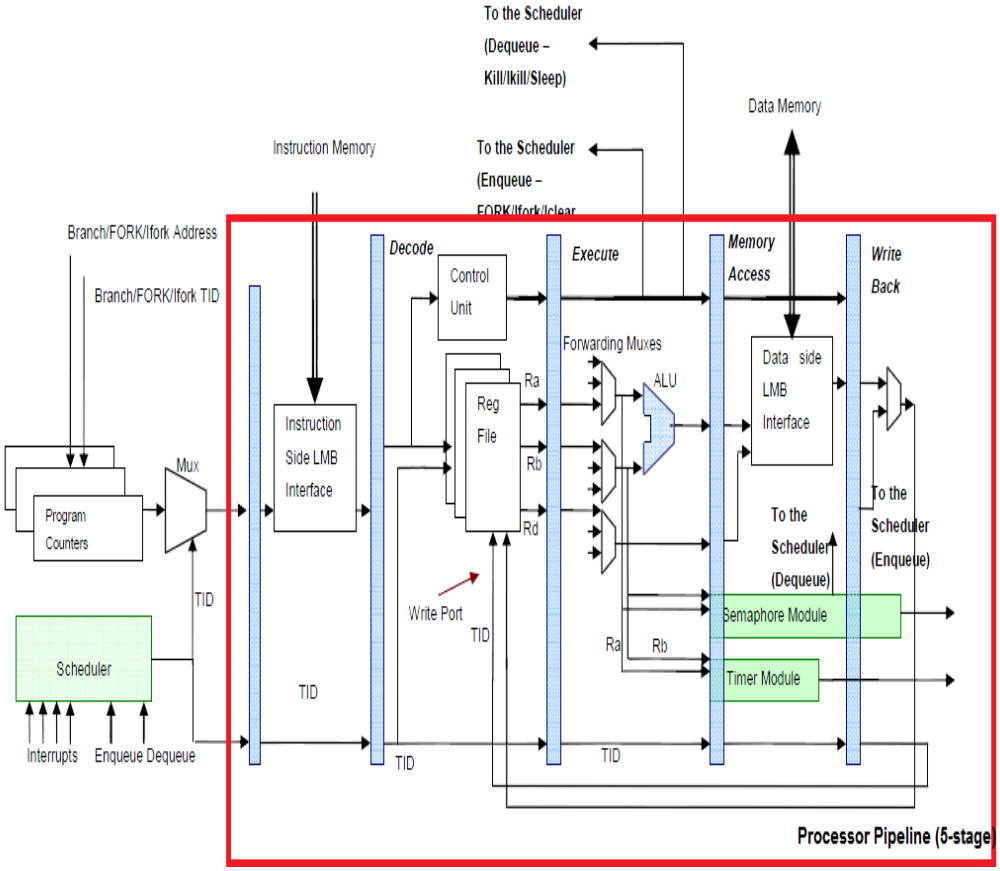


Figura 3: Processador Base com *tightly-coupled hardware* RTOS [4]

Na implementação do escalonador, Wijesenghe atribui a cada *thread* uma prioridade fixa que varia de 0 a 15, isto permite uma implementação simplificada do escalonador. O processador possui um banco de *Program Counter* que guarda para cada *thread* o respetivo endereço da próxima instrução. Os quatro sinais de interrupções são entradas do escalonador e correspondem as quatro *threads* de maior prioridade (12 a 15) permitindo-lhe efetuar a preempção destas. Os sinais de *enqueue e dequeue* são sinais de entrada e são fornecidas no estágio de execução por parte do processador e pelo módulo que implementa os *semaphore* no estágio de acesso a memória. Estes sinais permitem ao escalonador acrescentar, retirar ou posicionar *threads* na pilha de execução das tarefas consoante o tempo que estiveram a espera. A saída do escalonador existe o sinal 'TID' que é o identificador da *thread* a ser executada. O 'TID' é propagado ao longo de todo o *pipeline* do processador, para em caso de *hazard* este ser resolvido sem perder informação. Este sinal também é entrada do *multiplexer* à entrada do primeiro estágio, para fornecer a este ultimo, o valor do PC respetivo á *thread* a ser executada.

O módulo dos temporizadores em *hardware*, permite a cada *thread* ser colocada por ordem de espera para execução, no escalonador. Acontece que cada vez que uma *thread* é executada, o sinal de *enqueue* das restantes é atualizado para serem reordenadas de uma forma justa no escalonador. O módulo responsável por implementar os *semaphore* está incluído no estágio de acesso a memória do processador, com a finalidade de bloquear acessos não autorizados. Existem no total quatro *semaphore* em *hardware*, descodificando o valor do registo A, o módulo ativa o respetivo *semaphore* a ser usado.

Este exemplo retrata certamente a melhor implementação para remover o *overhead* do *context switch* e tornar o sistema embebido mais determinístico, mas tem a desvantagem de ser uma implementação menos flexível sendo exclusiva para este processador.

2.3.3 O ARPA-MT

O projeto ARPA-MT [1], desenvolvido por Arnaldo Oliveira, consistiu no desenvolvimento de um processador multitarefa com respetivo *hardware* RTOS (ver Figura 4). Este projeto é constituído por três módulos: o CPU, o Cop2-OSC e o Cop0-MEC.

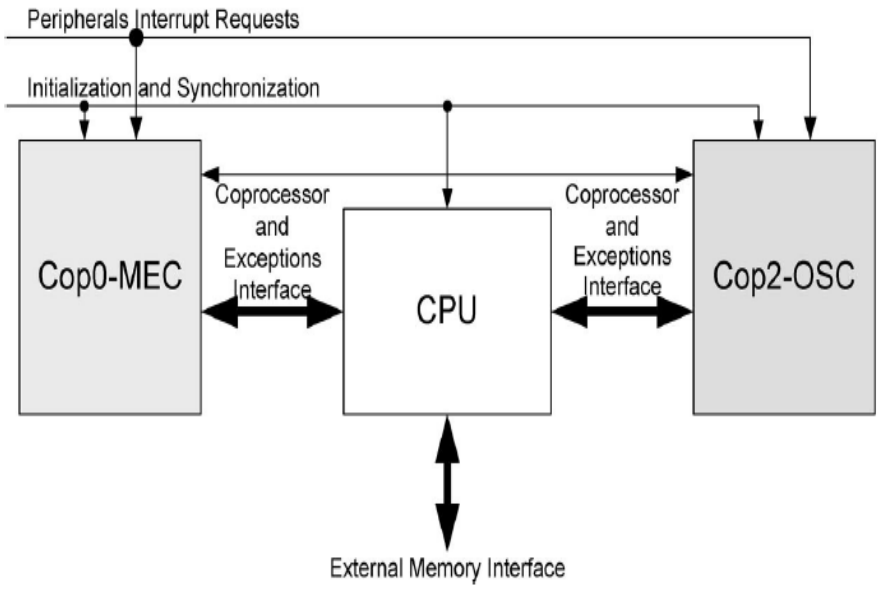


Figura 4: Visão geral do projeto ARPA-MT. CPU (MIPS32), Cop0-MEC (*memory managment unit*, configuração e *handling* interrupção e exceções), Cop2-OSC (*hardware* RTOS) [1].

O CPU contém a implementação standard da arquitetura do MIPS32 com cinco estágios de *pipeline*. O módulo Cop0-MEC é um coprocessador responsável por manipular e efetuar a gestão da memória, exceções e interrupções, enquanto o módulo Cop2-OSC é responsável por implementar as funcionalidades do *kernel* de um RTOS.

Este ultimo módulo é constituído internamente por um *pipeline*, que permite gerir todas as atividades relacionadas com gestão e deteção de *hazard*; um banco de registos de configuração e de dados, que permite ao utilizador configurar o comportamento do escalonador e verificar os dados guardados nos seus registos de controlo; uma unidade de relógio de tempo real, que gera os eventos periódicos na unidade de gestão das tarefas, de um modo similar ao projeto do RTBlaze [4]; uma unidade de gestão de tarefas, uma unidade de gestão de *semaphore* e para finalizar uma unidade de geração de exceções, que gera as exceções internas do módulo para depois as fornecer ao módulo Cop0-MEC. Estes módulos e suas interligações internas estão ilustrados na Figura 5.

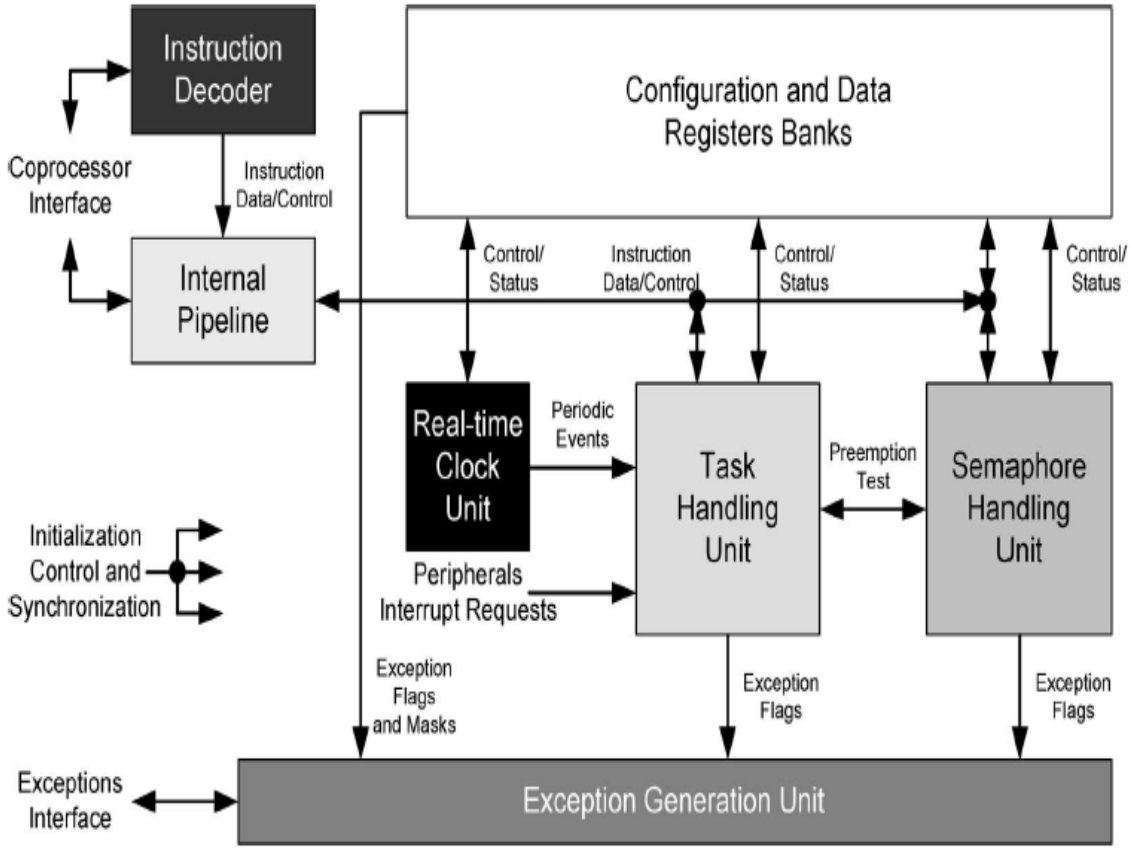


Figura 5: Diagrama de blocos internos do módulo Corp2-OSC [1]

Ao contrário do RTBlaze [4], o *hardware* RTOS desta implementação possui uma interface com o CPU, não se encontrando embutido no *pipeline* do processador. Sendo assim trata-se de um módulo *loosely-coupled* ao processador porque utiliza um barramento *standard*.

2.3.4 ARTESSO *hardware* RTOS

O projeto elaborado por Naotaka Maruyama, o ARTESSO, cujo principal componente é o *hardware* RTOS [11], foi criado tendo em vista libertar o CPU das instruções relacionadas com o sistema operativo de tempo real, das escritas para a memória e da tarefa de calcular o *checksum* para o protocolo TCP. Como se pode verificar na Figura 6, o CPU com o *tightly-coupled* RTOS comunica com os restantes módulos (memória de instrução e de dados, DMAs...) através de um barramento externo.

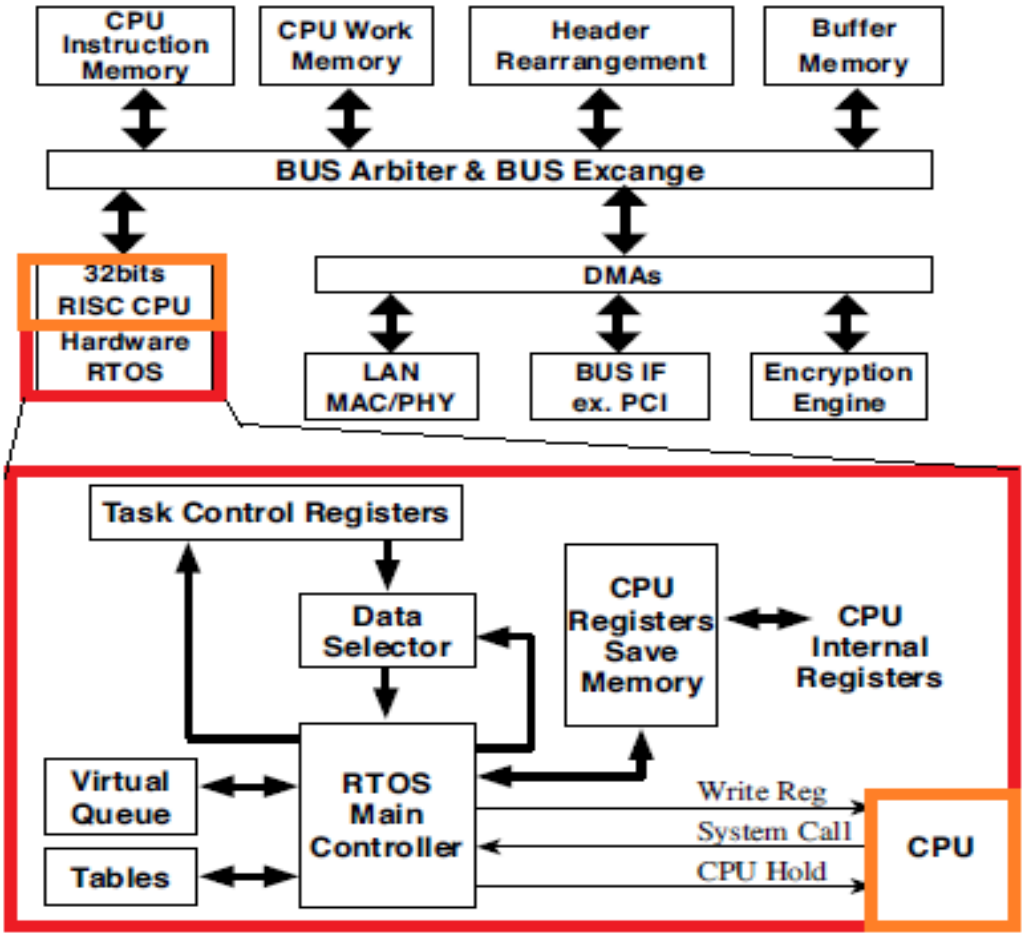


Figura 6: Configuração do ARTESSO, e arquitetura do *hardware* RTOS [11].

Esta é mais uma implementação em *hardware* de um RTOS, para libertar o CPU de processamento supérfluo, sendo que neste caso em concreto, esta libertação de carga de processamento é feita para que o CPU esteja totalmente dedicada ao processamento do protocolo TCP,

2.3.5 Conclusão

Com um olhar crítico nas implementações acima referidas, conclui-se que o tipo de acoplamento, entre os módulos a desenvolver e o processador principal é um aspeto importante a definir na elaboração deste projeto, pois um módulo *tightly-coupled* é mais robusto e eficiente, mas menos flexível enquanto um módulo *loosely-coupled* é mais fácil de migrar mas poderá não ser tão eficiente devido às perdas de tempo na comunicação com o processador.

Outro aspeto encontrado é que as migrações apresentadas são exemplos de módulos completos e não apenas de funcionalidades, isto acontece porque se torna mais vantajoso passar para *hardware* módulos mais completos, do que parcelas ou funções implementas em *software*.

Também é de salientar que em todas as implementações referidas anteriormente, o *hardware* acoplado consiste numa implementação personalizada das funcionalidades de um RTOS genérico e não provém da migração direta de funções ou estruturas de um RTOS comercial onde poder-se-á escolher a execução em *hardware* ou em *software*, como será o caso desta dissertação.

Capítulo 3

Tecnologias de suporte à migração

Após efetuar a análise geral do projeto e decidir quais o RTOS e o processador a utilizar, serão apresentadas neste capítulo, as características e a constituição destes, de uma forma aprofundada para se perceber o seu funcionamento.

3.1 O sistema operativo de tempo real – eCos™

O eCos™ é um sistema operativo de tempo real *open-source* que possui a característica de ser altamente parametrizável. Esta filosofia de implementação permite-lhe reduzir o seu tamanho quando um sistema embebido possui limitações nos seus recursos, por exemplo, a memória. Sendo assim, para sistemas embebidos minimalistas, o eCos™ pode ser compilado sem algumas funcionalidades, que geralmente são usadas como suporte de recursos avançados para sistemas embebidos mais complexos, reduzindo assim o seu tamanho na memória. Esta característica permite-lhe adaptar-se facilmente aos requisitos e restrições do sistema embebido que o acolhe.

Como é de esperar o eCos™ possui algumas funcionalidades que são esperadas quando estamos a falar de sistemas operativos de tempo real, isto inclui gestão de *threads,* escalonamento, sincronização entre *threads*¸ temporizadores, gestão de interrupções, gestão de exceções e *device*

drivers. O eCos™ fornece estas funcionalidades, divididas pelos seguintes componentes que constituem a sua arquitetura [11]:

- *Hardware Abstraction Layer (HAL).*
- *Device Drivers* – Inclui *drivers* para porta série, Ethernet, Memória Flash entre outros.
- O *kernel.*
- As bibliotecas ISO C, *math*, POSIX e μTron
- Suporte para o GNU *Debugger* (GDB) – disponibiliza o *software* necessário para comunicar com um *host* GDB com aplicação de *debug*.

Estes componentes estão divididos por quatro camadas como se pode verificar na Figura 7.

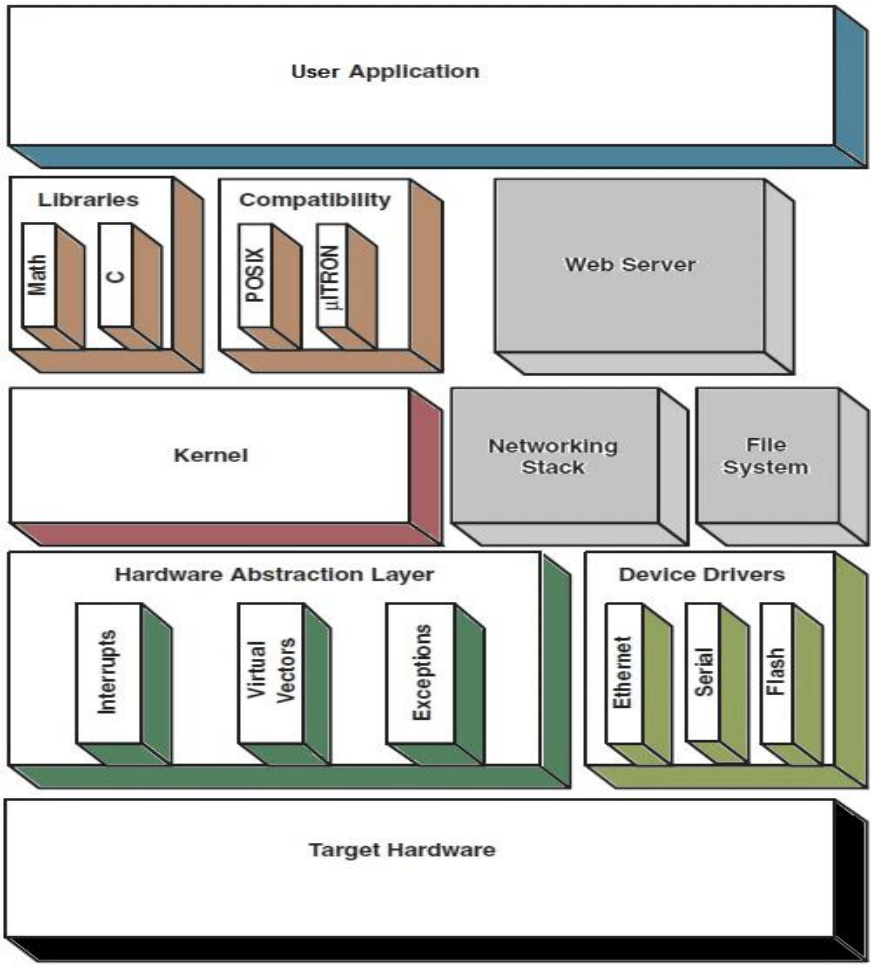


Figura 7: Exemplificação das camadas constituintes do eCos™ [12]

Para configurar e compilar o eCos™ é utilizada uma ferramenta o *Configtool*, sendo necessário o repositório do eCos™ e a *toolchain* respetiva para efetuar *cross-compile* para o processador escolhido (ver Anexo I e Anexo II).

Sendo o *kernel* um componente fulcral, assim como HAL que permite ao sistema operativo ter uma abstração do funcionamento do *hardware*, serão os dois componentes da arquitetura do eCos™ a serem aprofundados nos subcapítulos 3.1.1 e 3.1.2 respetivamente. Por fim em 3.1.3 será explicado a interação entre o HAL e o *kernel* para efetuar a gestão de exceções e interrupções.

3.1.1 *Hardware Abstraction Layer* (HAL)

A camada de abstração de *hardware* (HAL), presente na maioria dos sistemas operativos, é uma camada de *software* que permite isolar recursos dependentes da arquitetura do *hardware*, de modo a serem utilizados de uma forma simples através de API's. Desta forma, o HAL permite à camada da aplicação aceder ao *hardware*, por intermédio de API's do *kernel*. No entanto, este pode não ser o controlador de todo o *hardware* do sistema sendo necessário recorrer a *device drivers*.

O HAL do eCos™ está codificado em C e *assembly*, este está dividido em 3 módulos: A arquitetura, a variante e a plataforma. O primeiro módulo, a arquitetura, contém o código necessário para o arranque do CPU, a comutação de contexto, a entrega das interrupções e outras funcionalidades específicas ao ISA da família do processador utilizado. Como exemplo de famílias de processadores existe o OpenRISC, SPARC ou MIPS. A variante é para distinguir um processador específico pertencente a família de processadores escolhido, no caso de escolher uma arquitetura OpenRISC, uma variante possível é o OR1200 que pertence a esta família. Neste módulo está geralmente o código para o suporte de MMU e outros periféricos, que pertencem exclusivamente ou possuem uma alteração do seu acesso para a determinada variante. Por último, a plataforma, refere-se ao código de inicialização para um determinado *hardware* que inclui o processador escolhido ou uma variante, isto é, placas de desenvolvimento com FPGAs ou com o processador inserido fisicamente em ASIC.

Para entender melhor a estrutura do HAL e suas funções, os próximos pontos esclarecerão como este efetua a gestão de uma interrupção e de uma exceção e como é iniciado no arranque do

sistema. Começar-se-á por esclarecer como está constituída a diretoria do HAL, mais especificamente a diretoria do HAL para a arquitetura OpenRISC.

3.1.1.1 A diretoria do HAL e a arquitetura OpenRISC

A nível de diretoria, o HAL do eCos™ está dividido pelas diversas arquiteturas de processadores, no entanto, para evitar repetição de código e erros no *call* das funções, uma diretoria "*common*" possui o código que é semelhante para todas as arquiteturas e as macros que irão chamar as funções respetivas. Dentro de uma arquitetura existe três sub-diretorias diferentes, uma diretoria "*arch*" que possui o código específico para essa arquitetura, que é semelhante para todas as variantes e plataformas escolhidas da mesma arquitetura, uma diretoria "sim", caso haja um simulador para esta arquitetura de processador, e por último uma diretoria para cada um das variantes ou plataformas dessa família, com o seu código específico no interior. Na Figura 8, está representado de uma forma simples como estão encapsuladas as diversas pastas e subpastas do HAL.

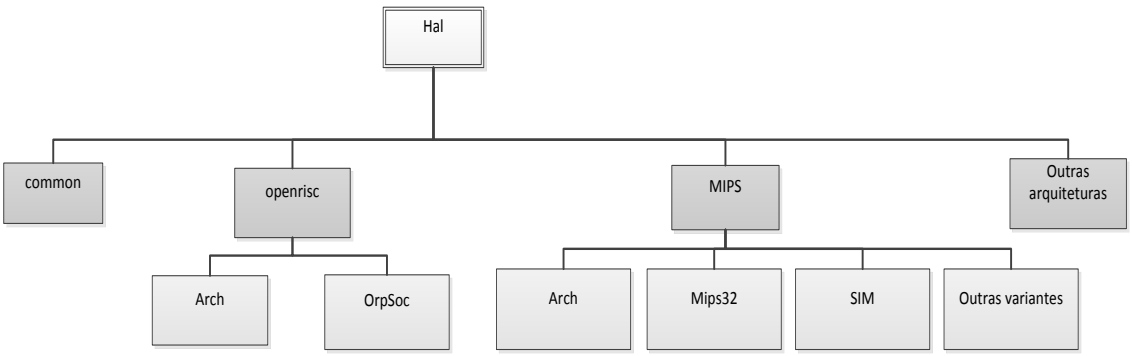


Figura 8: Exemplificação da estrutura da diretoria do HAL do eCos™.

Revelada a distribuição do HAL, mencionar-se-á para o caso da arquitetura OpenRISC quais são os ficheiros e as funções/serviços disponibilizados para efetuar uma migração sem descaracterizar a estrutura do HAL. Na Figura 9, está ilustrada a composição do HAL do eCos™ para a arquitetura OpenRISC.

No interior do diretório do HAL para a família OpenRISC existe duas diretorias: "*include*" com os *header files* e uma diretoria "*src*" que contém o código fonte. A diretoria "*include*" tem os *header files* ".h" com as macros que definem:

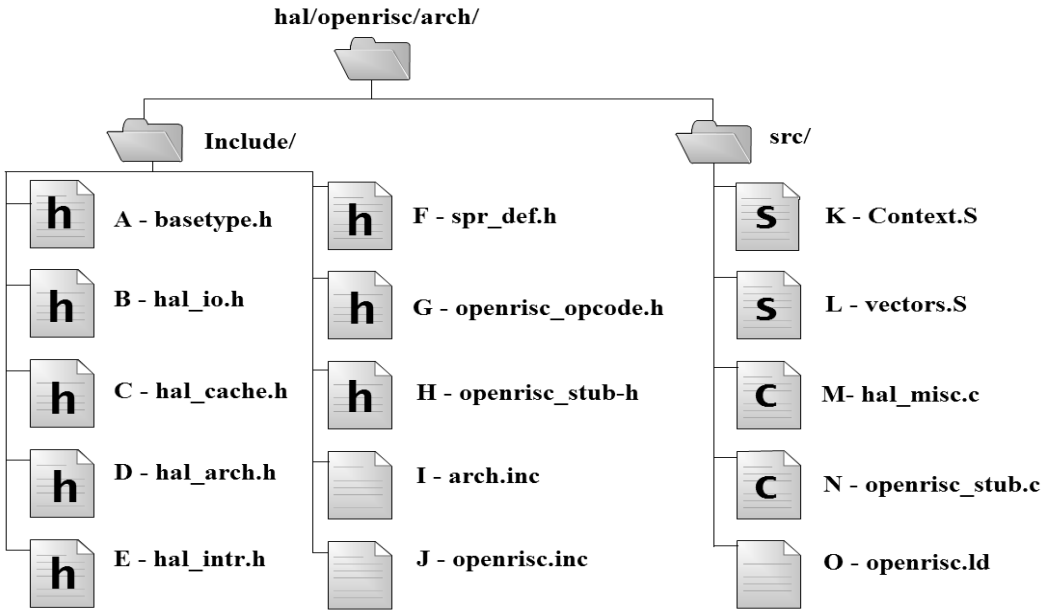


Figura 9: Constituição do HAL para a arquitetura OpenRISC

A. "basetype.h" – o tipo de máquina, neste caso, que se trata de uma máquina '*Big Endian*'.

B. "hal_io.h" – as APIs para manusear o registo de controlo dos pinos I/O, para aceder-lhes individualmente ou por registos de 8, 16 ou 32 bits.

C. "hal_cache.h" – as APIs para acesso e controlo das caches de memória, onde também está definido o tamanho total das caches (4096 bytes) e o tamanho de bytes por linhas (16).

D. "hal_arch.h" – a abstração do *hardware* no que diz sentido ao acesso aos registos especiais de propósito geral, a manipulação de bit como a "*flag*" e a definição do tamanho das pilhas para interrupção e para as *threads*.

E. "hal_intr.h" – o suporte para as interrupções e para o temporizador/contador, estão também definidas os vetores de Interrupção assim como as APIs para ativação ou desativação destas.

F. "spr_def.h" - o valor de cada bit dos *Special Purpose Register*, para serem endereçados corretamente.

G. "openrisc_opcode.h" – os formato das instruções de salto e os respetivos *opcodes*.

H. "openrisc_stub.h" – as APIs de suporte ao *debugger gdb*, para obter valores de registos, criação remoção de *breakpoints* entre outros.

Os *header files* ".inc" para serem usados conjuntamente com os ficheiros em Assembly, também se encontram nesta diretoria. Estes definem:

I. **"arch.inc"** – as macros da inicialização de funções em *assembly*, o valor do *stack pointer* (r1), do *frame pointer* (r2) e do tamanho dos registos.

J. **"openrisc.inc"** – do mesmo modo que o "hal_arch.h" a arquitetura, mas para o código em *assembly*.

A pasta "src" contém o código fonte onde as APIs estão implementadas. Os ficheiros em *assembly* "contexto.S" e "vectors.S" contêm as seguintes funções:

K. **"context.S"** Contém as funções "hal_thread_switch_context", "hal_setjmp", "hal_thread _load_context" e "hal_longjmp". A função "hal_thread_switch_context" realiza como o seu nome indica a comutação de contexto, isto é, salvaguarda o estado de uma *thread* e carrega a informação da próxima *thread* a ser executada. A função "hal_setjmp" e "hal_longjmp" que salvaguardam e carregam respetivamente, os registos de prepósito geral antes de efetuar um salto. Por fim está a função "hal_thread_load_context" que assegura o carregamento correto da informação da *thread* nos respetivos registos.

L. **"vectors.S"** – Contém as funções "start", "hal_default_exception_vsr", "hal_default_ interrupt_vsr", "hal_interrupt_stack_call_pending_DSRs". A função "start" é o ponto de partida para a inicialização do *hardware* no arranque do sistema. A função "hal_default_exceprion_vsr" chama a função C "hal_exception _handler" que trata de efetuar a manipulação da exceção. A função "hal_default_interrupt_vsr" faz o manejo das interrupções com o escalonador bloqueado. Por fim a função "hal_interrupt_stack_call_ pending_DSRs" faz o manejo de interrupções mais complexas ativas, com o escalonador e as interrupções ativas, segundo a sua prioridade.

Os ficheiros em C, "hal_misc.c" e "hal_stubs.c" contêm as seguintes funções:

M. **"hal_misc.c"** – Este ficheiro C contém todas as funções do HAL, algumas destas estão implementadas, outras efetuam o *call* de funções implementadas em assembly, apresentadas nos ficheiros anteriores.

N. **"openrisc_stubs.c"** – Este ficheiro contém todas a implementação das APIs para o uso do *debbuger gdb*.

O último ficheiro, **"openrisc.ld"** é um *linker script* para a arquitetura OpenRISC, para definir o local da VSR (*Vector Service Routine*).

3.1.1.2 Inicialização do HAL

A inicialização do *hardware abstraction layer* no arranque do sistema passa por várias etapas, pois todo o *hardware* é preparado nesta fase, para ser utilizado posteriormente. A Figura 10 representa as diversas etapas da inicialização do HAL.

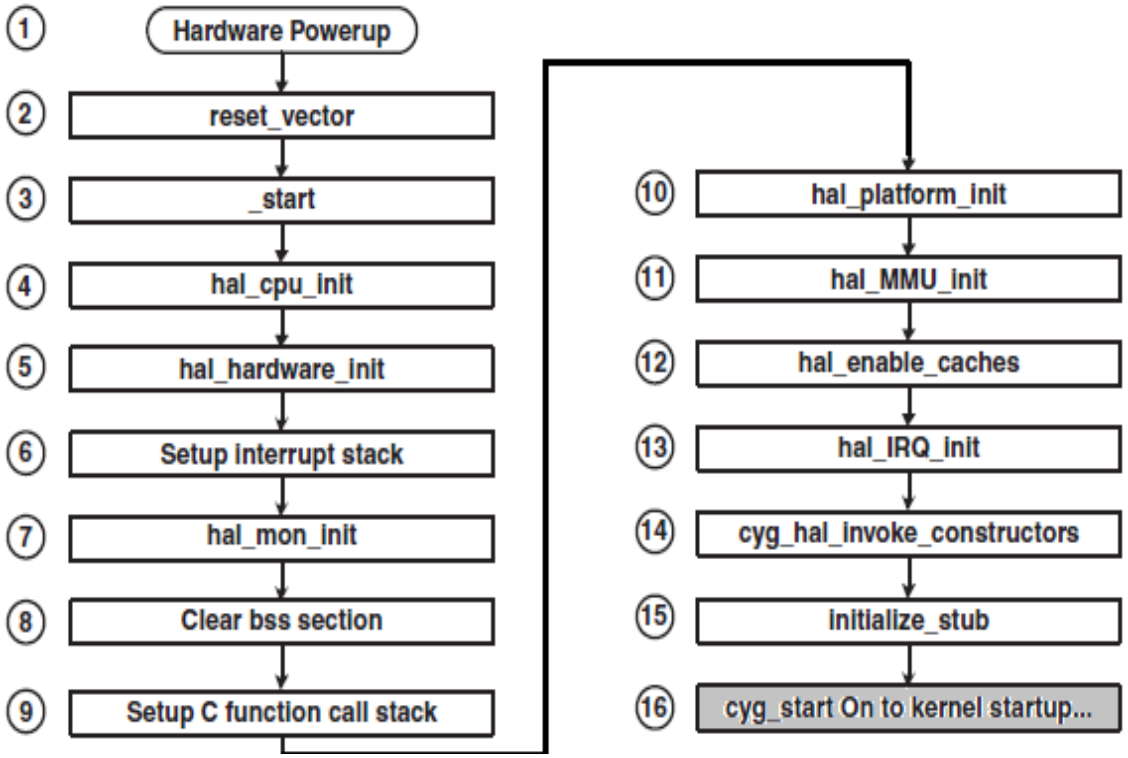


Figura 10: Etapas da inicialização do HAL [12]

Numa primeira fase o *hardware* é ligado: quando acontece, o *Program Counter* (PC) aponta para o endereço de *reset*, que no caso do OpenRISC é 0x100. Após o *reset* ser efetuado a função "start" é iniciada (presente no ficheiro Vector.S). Esta função trata-se da função principal para a inicialização do HAL, pois as seguintes funções presentes na Figura 10 (de ④ a ⑯) são funções chamadas dentro da função "start" ③.

Assim que a função "start" arranque a função "hal_cpu_init"④ é inicializada, está é responsável por preencher os registos de prepósito geral com valores iniciais, também é responsável por desativar a cache de instruções e dados de modo a que nenhuma informação errada passe para o *pipeline* do processador. O próximo passo é a execução de "hal_hardware_init"⑤: esta função é responsável por inicializar o *hardware* específico à variante escolhida. Configura a cache, coloca os

registos de controlo de interrupção com valores por defeito e configura os registos de *chip-select* relativos à variante (esta ultima parte é realizada se for escolhida uma variante, caso não seja não é realizada). A próxima rotina denominada "*Setup_interrupt_stack*" ⑥reserva uma área específica para guardar as informações do processador quando se realiza uma interrupção. De seguida é lançada a função "*hal_mon_init*"⑦ que é responsável por instalar a tabela de *Vector Service Routine* por defeito, para o caso do OpenRISC é entre a posição 0x200 a 0xf00. A função em ⑧ trata de limpar a secção onde as variáveis estáticas estão alocadas e a tarefa posterior ⑨ cria uma pilha, onde as funções em C poderão ser chamadas pelo código Assembly presente no ficheiro "Vector.S". A próxima etapa ⑩ complementa a tarefa realizada em⑦, ao acrescentar novas exceções referentes à plataforma escolhida. Em ⑪ as *Memory Managment Unit* responsáveis por traduzir endereços lógicos para físicos e desencadear mecanismos de proteção de acesso as caches, é inicializada. A função "*hal_enables_caches*" ⑫ ativa as caches de instrução e de dados. Em ⑬ é ativado o módulo responsável por gerir as interrupções externas. Na etapa ⑭ a função "*hal_invoke_constructors*" chama os construtores C++ das diferentes classes, que são necessárias para ao *kernel*. Seguidamente em ⑮ se a opção for selecionada, a função "initialize_stub" instala as *trap handler* responsáveis por gerir as ações de *debug* e inicializa o módulo de *debug*. Para finalizar a inicialização do HAL em ⑯, a função "cyg_start" passa o controlo ao *kernel* para este efetuar a sua inicialização.

3.1.2 O *kernel*

O *kernel* é o núcleo do sistema operativo eCos™; é responsável por fornecer as funcionalidades de um RTOS, como escalonamento, gestão e sincronização entre *threads* mas também com o auxílio do HAL, faz a gestão de exceções e interrupções (ver subcapítulo 3.1.3).

A camada de aplicação tem a particularidade das suas APIs não devolver valores, isto porque o tratamento destes valores consome muito processamento por vezes desnecessário. Devido a esta opção de implementação, o *kernel* é impossibilitado de recuperar de erros por si só, parando a aplicação quando um erro surge. No entanto o eCos™ disponibiliza *assertions* que podem ser habilitados ou desabilitados no momento de configuração do RTOS. *Assertions* são um suporte na

hora de efetuar o *debug* do sistema, pois caso um erro surja a *assertion* informa o utilizador do tipo de erro que aconteceu efetuando um *prompt*, para posteriormente ser corrigido.

Sabendo agora como está composto o *kernel* do eCos™ os subcapítulos 3.1.2.1 a 3.1.2.3 vão de uma forma pormenorizada explicar a composição e o funcionamento das suas funcionalidades, para finalizar com o *kernel* do eCos™ em 3.1.2.4 será exemplificada a sua inicialização.

3.1.2.1 Escalonadores e controlo do escalonamento

O escalonador é a peça principal no *kernel* de um sistema operativo pois é este que efetua o trabalho de determinar qual a próxima *thread* a ser executada, fornece os mecanismos de controlo e sincronização entre *thread* e controla o efeito das interrupções na execução de uma *thread*. No entanto há que ter em atenção que este pode ser impedido de efetuar um ponto de escalonamento sendo bloqueado por uma interrupção, ou até mesmo por uma *thread* com a função "cyg_scheduler_lock"*.*

Quando um ponto de escalonamento surge é efetuada uma comutação de contexto. As comutações de contexto são eventos em que é parada a execução de uma tarefa e iniciada a execução de outra tarefa. Quando tal acontece o estado do CPU e a informação sobre áreas de memória atribuídas à *thread* que estava a executar são guardados, e as informações relativa à *thread* pronta a ser executada são carregadas. Ora no momento de troca de contexto, o CPU não está a executar nenhum código relativo à aplicação, surgindo aqui um *overhead.* Sendo assim, na hora de escolher o escalonador, é necessário ter em mente as necessidades da aplicação e o possível *overhead* da comutação de contexto que pode surgir.

O eCos™ fornece dois tipos de escalonadores, o "Multilevel queue" e o "Bitmap", sendo que a escolha de um, só pode ser feita em detrimento de outro.

O escalonador "Multilevel queue" permite a execução de múltiplas *threads* com a mesma prioridade. As prioridades vão de '0' e podem chegar, dependendo da configuração, até '31', sendo que a *thread* com o menor número tem maior prioridade. Este escalonador permite efetuar *preemption,* isto é, se o escalonador o achar necessário pode interromper a execução de uma *thread* e colocar em execução uma *thread* com maior prioridade que esteja pronta para ser executada. Também possui a capacidade de *timeslicing,* isto é, o escalonador consegue dividir o tempo de execução para cada *thread* com o mesmo grau de prioridade, permitindo a cada *thread*

um tempo de execução. A Figura 11 ilustra o funcionamento do escalonador "Multilevel queue" com preempção da *thread* C e *timeslice* entre a *thread* A e B.

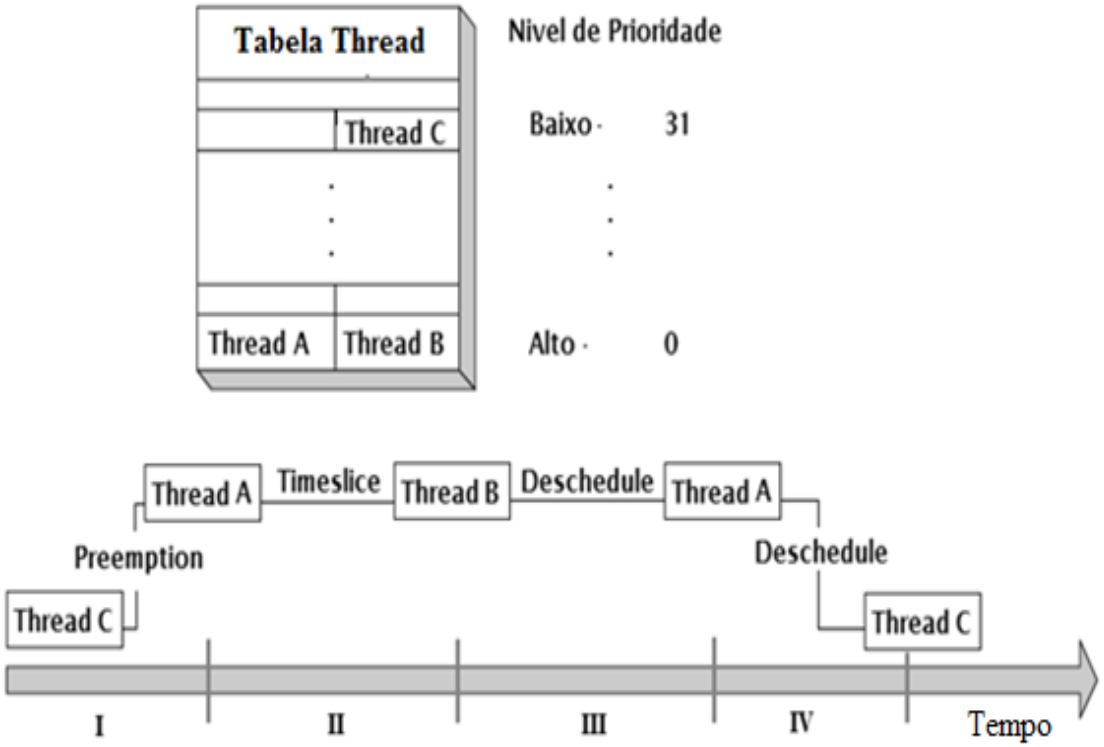


Figura 11: Exemplo do escalonamento operado pelo escalonador *Multilevel queue*.

O escalonador "Bitmap" é mais simples que o anterior, permite preempção mas não existe *timeslice* (como em II na Figura 11), pois não precisa desta característica visto que este escalonador só permite uma *thread* por prioridade.

A Figura 12 apresenta o diagrama de classes para a implementação dos escalonadores no eCos™, neste caso refere-se ao Bitmap. A classe "Cyg_Scheduler_Base" trata-se de uma classe base para ambos os escalonadores cuja subclasse "Cyg_Scheduler_implementation" herda todos os atributos e métodos. A classe "Cyg_Scheduler_Implementation" pode apresentar variações dependendo do tipo de escalonador que escolhemos (*Multilevel queue* ou *Bitmap*), pois esta apresenta os métodos e atributos específicos ao escalonador. Esta classe possui dois atributos, "run_queue", que é a lista ligada de *threads* prontas a serem executadas, e "thread_table", que é um apontador para a tabela onde estão registadas todas as *threads* por prioridade. Por fim a

subclasse "Cyg_Scheduler" herda todos os atributos e métodos das duas classes anteriores, fornecendo a abstração da implementação do escalonador perante todo o sistema operativo.

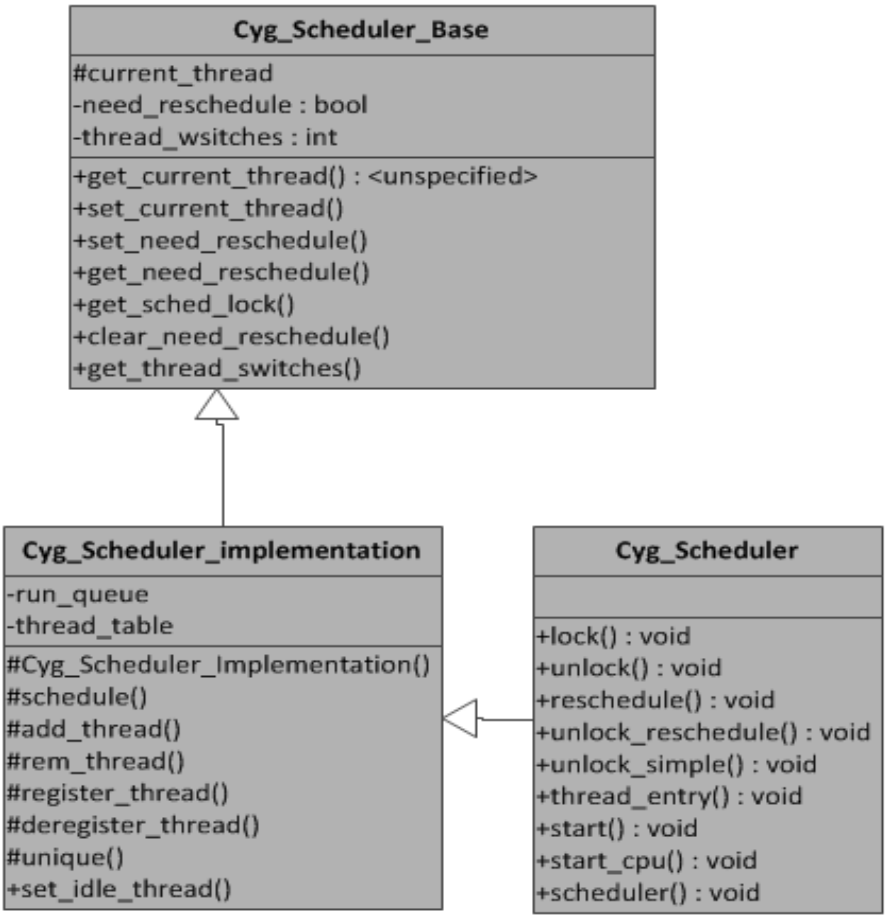


Figura 12: Diagrama de classes - escalonador *Bitmap*

3.1.2.2 Gestão de *threads*

Uma *thread*, no contexto de sistemas operativos, consiste numa linha ou encadeamento de execução única. A Figura 13 ilustra como é constituída a classe "Cyg_Thread" que fornece a implementação dos métodos e os atributos próprios da *thread.* "Cyg_Thread" herda atributos e métodos de duas classes bases que lhe dam a abstração a nível do *hardware* e do escalonador.

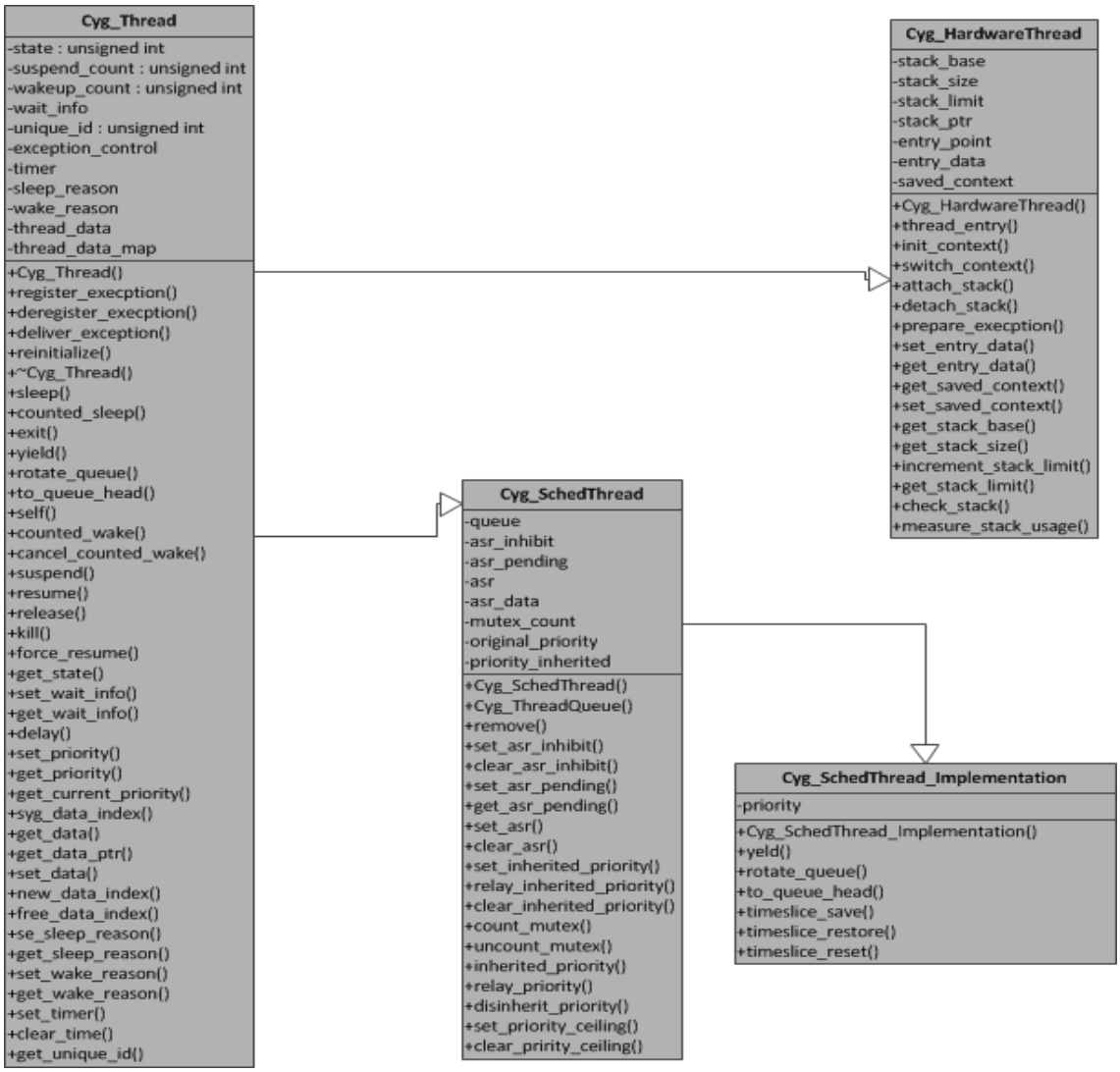


Figura 13: Diagrama de classes - *threads*

"Cyg_hardwareThread" dá a abstração a "Cyg_Thread" em relação ao *hardware*, fornecendo-lhe os atributos e métodos necessários para efetuar operações a nível do *hardware*, relativas a *thread*. Nestes atributos herdados, constam os apontadores para a pilha da *thread*: esta pilha é disponibilizada pela aplicação e fornece um espaço reservado de memória para guardar variáveis locais.

A classe "Cyg_SchedThread" é uma subclasse que herda da classe base "Cyg_SchedThread_ Implementation", os atributos e os métodos relativo ao tipo de escalonador escolhido e fornece a "Cyg_Thread" a abstração a nível do escalonador. Possui um atributo "queue" que é o apontador para a lista de espera de *threads*, para serem executadas. Este apontador fornece esta informação no tipo de classe abstrata "Cyg_ThreadQueue", Figura 14, que possui APIs para remover,

incrementar ou decrementar a posição das *threads* na lista de espera, assim como um atributo, herdado por "Cyg_ThreadQueue_Implementation", que é a lista ligada de *threads* no estado *wait*.

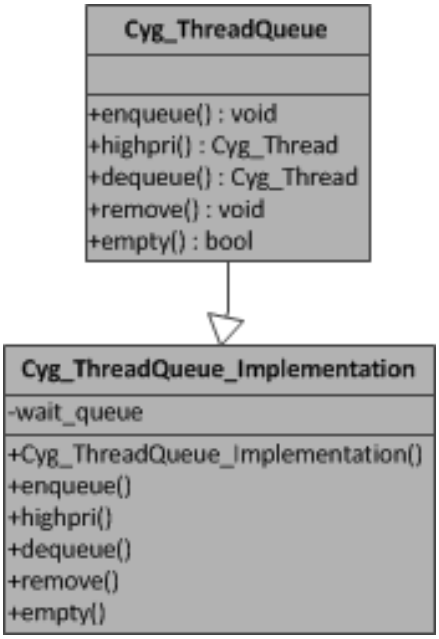


Figura 14: Diagrama de classes – lista de *threads*

No decorrer de uma aplicação uma *thread* pode ter diversos estados (guardado no atributo "state", classe "cyg_thread" na Figura 13), esses estados no eCos™ podem ser *running, sleeping, suspended, creating* e *exited.*

Para entender como é efetuada esta troca, usando o exemplo de uma *thread* 'TA' que ainda não está criada. Quando 'TA' é criada é registada na tabela de *threads* e inserida na lista ligada de *threads* em espera, com o estado "sleeping". Quando 'TA' possuir todos os recursos necessários a sua execução, esta é removida da lista de espera e é inserida na lista de *threads* pronta a ser executada, com o estado "suspended". Caso aconteça um ponto de escalonamento e 'TA' estiver no topo da fila, ela passará a ser executada (estado "running"). Enquanto 'TA' estiver a ser executada, se uma *thread* de maior prioridade estiver a espera na fila de execução, o escalonador efetua uma preempção, e recoloca 'TA' num estado "suspended". Por outro lado se 'TA' necessitar de recursos que não possui para continuar a sua execução, ela é retirada do estado "running" e recolocada na fila de espera, com o estado "sleeping". Caso o regresso à lista de espera tenha sido causado por esta ter terminado toda a sua execução, TA é "exited" e removida da tabela e da lista. A Figura 15 mostra como é efetuada a gestão das *threads* no eCos™

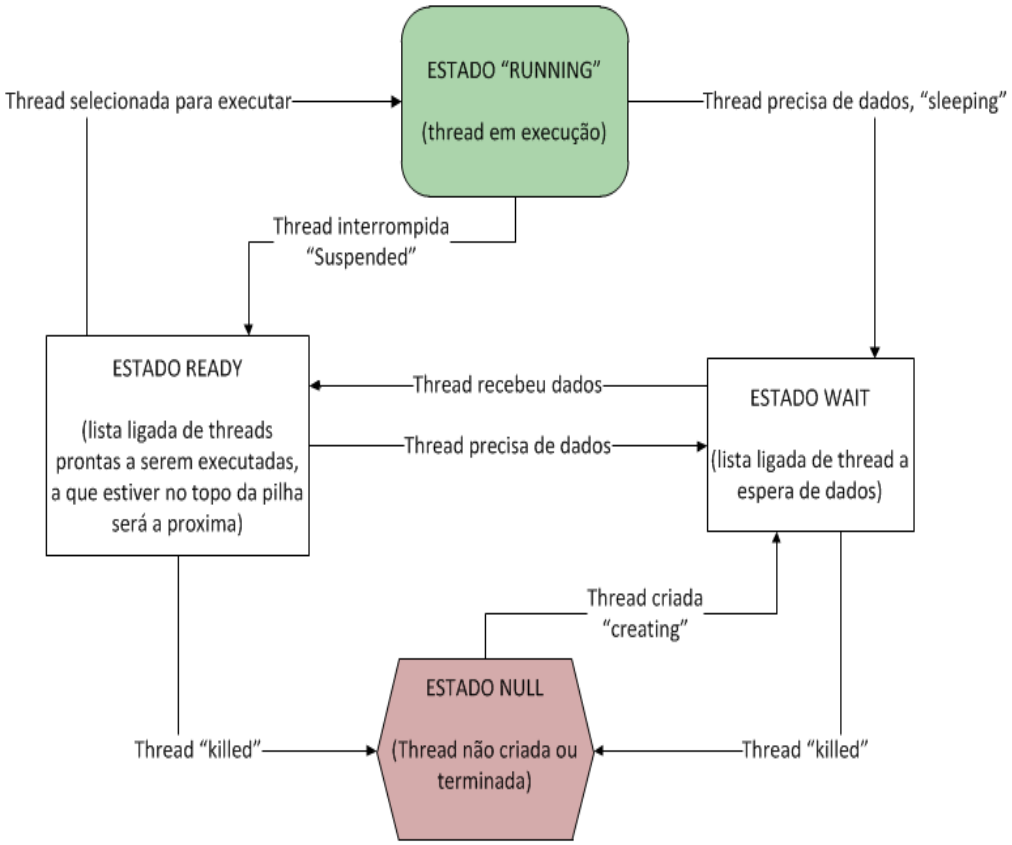


Figura 15: Gestão das *Threads*

3.1.2.3 Mecanismos de sincronização e comunicação entre *threads*

Os mecanismo de sincronização entre *threads* são ferramentas indispensáveis quando é necessário sincronizar fluxos de execução e/ou partilhar recursos entre as *threads*, evitando assim as *race conditions.* O eCos™ fornece quatro mecanismos de sincronização e comunicação diferentes, *Mutexes (A), Semaphore (B), Conditions variables (C) e Flags (E).*

3.1.2.3.1 Mutexes

O *Mutex* (*Mutual Exclusion object*) é um mecanismo de sincronização que possui apenas dois estados, bloqueado e desbloqueado. Quando uma *thread* possui o *mutex*, isto é, quando exerce o bloqueio do mutex usando a função "cyg_mutex_lock", esta será a única que poderá desbloqueá-lo usando "cyg_mutex_unlock", sendo que, se outra *thread* estiver a espera para bloquear o *mutex* está terá que esperar que a primeira o liberte.

Este mecanismo de sincronização também possui a particularidade de fornecer dois tipos de proteção contra o efeito de *priority inversion*. O efeito de inversão de prioridade, 'A' na Figura 16, acontece quando, uma *thread* com prioridade alta não pode continuar a executar porque necessita de bloquear um *mutex,* bloqueado por uma *thread* de baixa prioridade e que posteriormente, uma *thread* de prioridade média seja executada em detrimento da que possui o *mutex,* porque tem prioridade em relação a esta.

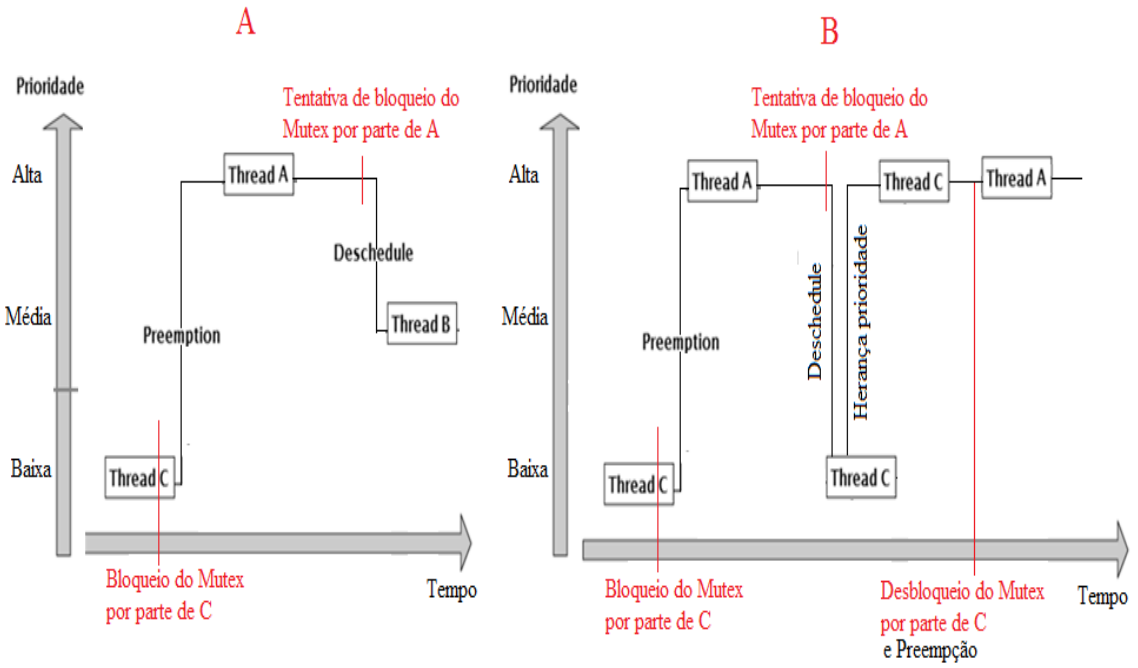


Figura 16: Efeito de *priority inversion* (A) e *priority inheritance protocol* (B)

A primeira proteção contra a inversão de prioridade denomina-se *priority ceiling protocol* (PCP), em que é dada ao *mutex* um certo valor de prioridade e a *thread* ganha essa prioridade enquanto possuir este *mutex*. No entanto este mecanismo de proteção não funciona por si só e o programador necessita de determinar quais e como atribuir as prioridades aos *mutexes*, para evitar a inversão de prioridade.

A segunda proteção contra este efeito indesejável, 'B' na Figura 16, é denominada de *priority inheritance protocol* (PIP). O PIP funciona da seguinte forma, quando uma *thread* de menor prioridade possui um *mutex* e uma *thread* de maior prioridade esteja a espera deste, a *thread* de menor prioridade herda o nível de prioridade da *thread* que está a espera e executa até libertar o *mutex*.

3.1.2.3.2 Semaphore

Um *semaphore* é um mecanismo de sincronização, que contém uma contagem interna que indica se um recurso está bloqueado ou disponível. Existem dois tipos de *semaphore, counting semaphore*, e *binary semaphore*.

Binary semaphore são semelhantes aos *counting semaphore*, no entanto, a sua contagem apesar de ser incrementada nunca ultrapassa o valor de um (valor Binário 0 ou 1). Sendo assim, *binary semaphores* só possuem dois estados, bloqueados ou desbloqueados.

Couting semaphores possuem vários estados, dependendo do seu valor inicial de contagem. Ao criar um *sempahore,* o contador é inicializado com um valor positivo que indica o número de *threads* que o *sempahore* deixa passar antes de começar a bloquear o acesso ao recurso. Quando uma *thread* efetua uma chamada a função "cyg_sema_wait" o RTOS verifica se o contador é maior que zero; se for o caso, decrementa o contador de uma unidade e a *thread* continua a execução, caso contrário a *thread* fica bloqueada. Ao concluir o acesso ao recurso a *thread* efetua a operação "cyg_sema_post" e o contador é incrementado de um valor.

3.1.2.3.3 Conditions Variables

As *conditions variables* são usadas com *mutexes* e permitem que múltiplas *threads* acedam aos dados partilhados. Tipicamente existe uma única *thread* produzindo os dados, e uma ou mais *thread* à espera que os dados estejam disponíveis. A *thread* responsável pelos dados pode sinalizar uma única *thread*, ou várias, para passar para o estado de prontas a executar quando os dados estão disponíveis. As *threads* em espera podem então, consoante a sua posição na fila, processar os dados conforme a necessidade.

3.1.2.3.4 Flags

As *flags* são mecanismos de sincronização representados por um *word* de 32 bits. Cada bit na *flag* representa uma condição, o que permite que uma *thread* espere por uma das 32 condições ou de uma combinação de condições. Existem dois tipos de *threads*, as que esperam pela condição e as que ativam os bits de condição. As *threads* que estão à espera das condições podem esperar por uma condição específica, ou uma combinação de condições a serem realizadas, antes de poderem

começar a executar. A *thread* que efetua a sinalização pode então definir ou redefinir bits, de acordo com as condições específicas, de modo que a *thread* adequada possa ser executada.

3.1.2.4 Inicialização do *Kernel*

O *kernel* do eCos™, antes de lançar o escalonador e deixar iniciar a aplicação, tem que perfazer algumas operações. Estas operações só são realizadas se o HAL já tiver sido inicializado. A função "cyg_start" é a função principal na inicialização do *kernel*, esta chamará posteriormente as restantes funções apresentadas na Figura 17.

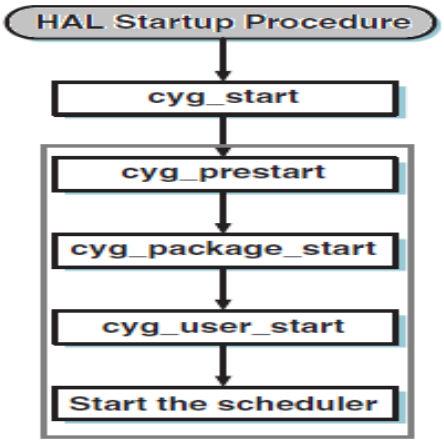


Figura 17: Inicialização do *kernel* do eCos™

A função "cyg_prestart" é uma função por defeito que não efetua qualquer tarefa, no entanto pode ser usada para inicializar partes do sistema antes de efetuar o carregamento das bibliotecas. Assim que a função "cyg_prestart" termina é lançada a função "cyg_package_start": esta inicializa as bibliotecas de compatibilidade tais como ISO C, *math,* POSIX ou μTRON. De seguida é chamada a função "cyg_user_start", onde são criadas as *threads*, mecanismos de sincronização, alarmes e rotinas de gestão de interrupções para a aplicação. Para finalizar é lançado o escalonador, após "cyg_user_start" ter terminado.

É de salientar que em qualquer altura o utilizador pode reescrever as funções de inicialização do *kernel* apresentadas na Figura 17, pois estas são fornecidas como APIs que podem ser usadas na codificação de um programa. No entanto, estas só serão executadas no momento de inicialização do *kernel*.

3.1.3 Gestão das Exceções e Interrupções

Uma exceção é um evento interno que ocorre durante a execução de uma *thread* que provoca um distúrbio no decorrer normal das instruções. Caso as exceções não sejam processadas atempadamente durante a execução do programa, falhas graves podem acontecer. É por isso que a gestão das exceções é importante, porque aumentam a robustez do sistema. Estas exceções podem ocorrer no sistema devido a um erro no acesso a memória ou até por um erro causado por uma operação de divisão por zero.

O HAL do eCos™ utiliza uma tabela de *Vector Service Routine* (VSR) para todas as exceções do sistema. A tabela VSR consiste num conjunto de apontadores que apontam para as secções de código respetivo. Sendo assim, quando ocorre uma exceção o processador verifica na tabela VSR qual foi ativa, para executar a sua respetiva rotina. A tabela VSR localiza-se num local fixo da memória e o HAL do eCos™ assegura a sua inicialização ao efetuar o *startup*. A Tabela 4 representa a exceções por defeito e os respetivos endereços na memória.

Tabela 4: Tabela VSR do eCos™ para o OpenRISC

Exceção – Vetor	Endereço	Exceção – Vetor	Endereço
Dummy Vector	0x000	*External interrupt*	0x800
Reset	0x100	*D tlb miss*	0x900
Bus_error	0x200	*I tlb miss*	0xa00
Data page fault	0x300	*Range*	0xb00
Instruciton Page fault	0x400	*Syscall*	0xc00
Tick timer	0x500	*Reserved*	0xd00
Unlaigned access	0x600	*Trap*	0xe00
Ilegal instruction	0x700		

As exceções são lidadas conjuntamente entre o HAL e o *kernel*. Inicialmente o HAL é responsável por efetuar a paragem na execução das instruções e obter o endereço da VSR na tabela. De seguida o *kernel* assume o controlo e é efetuada, através da função "cyg_hal_exception_handler", a gestão da exceção ocorrida. Depois este *handler* entrega-a ao nível da aplicação com "cyg_hal_deliver _exception", onde está instalada uma rotina que efetua um processamento suplementar à gestão da exceção, apresentando também as informações sobre a exceção ocorrida. Posteriormente é devolvido ao HAL a gestão da exceção e este finaliza a operação, restaurando o sistema onde foi interrompido. O exemplo, ilustrado na Figura 20 apresenta a gestão de uma exceção, mais particularmente quando um "system call" é realizado.

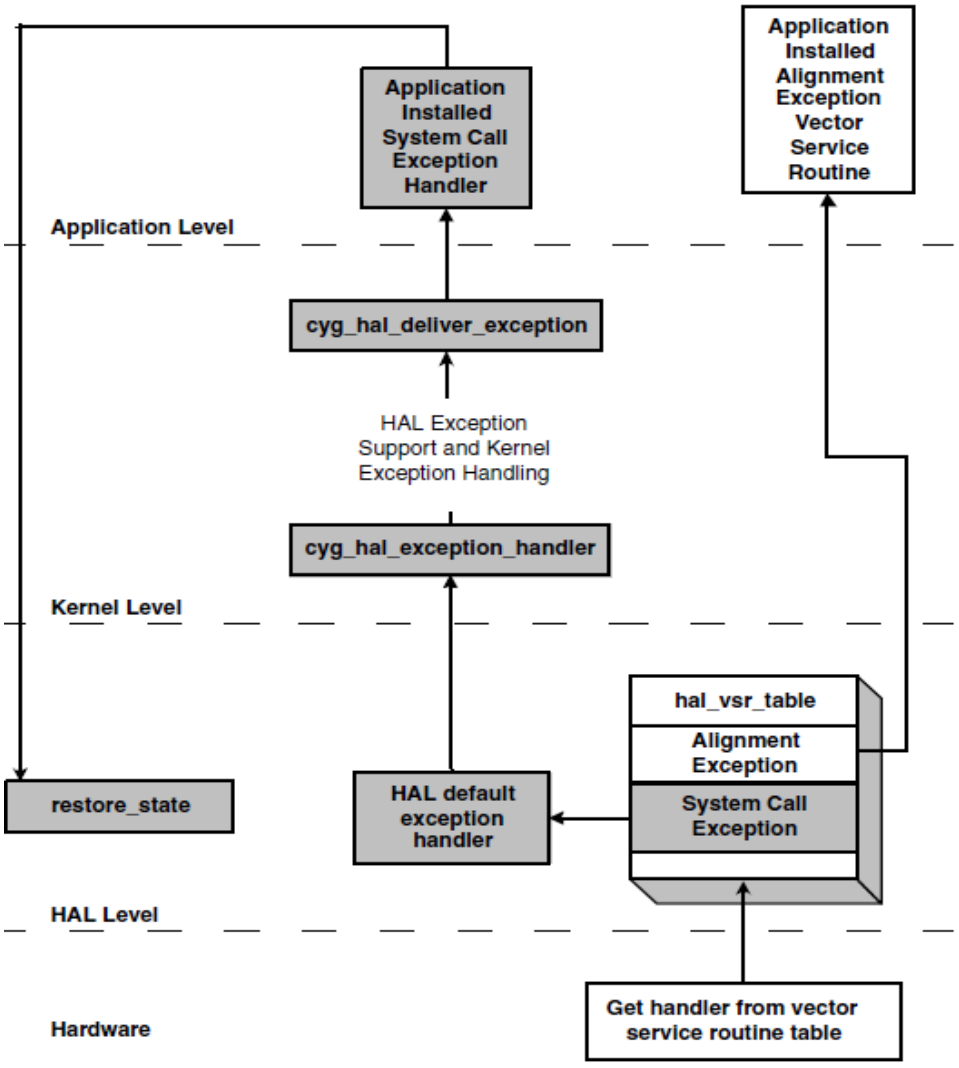


Figura 18: Gestão de uma exceção pelo eCos™ [12]

As interrupções são eventos assíncronos externos, que quando ocorrem provocam uma paragem da execução do programa. Quando ocorrem, o processador efetua um salto para o endereço da ISR (*Interrupt Service Routine*), dado pela tabela VSR (refere-se a exceção "external_interrupt") após descodificar qual a interrupção no conjunto de interrupções instaladas executa o código relativo a interrupção dada.

Um fenómeno critico a ter em conta é a latência no atendimento da interrupção. Este é o tempo perdido entre a ocorrência da interrupção e o momento em que é executada a sua ISR. O eCos™ divide então a gestão da interrupção em duas partes, em que a primeira é feita pela ISR e a segunda pela DSR (*Deferred Service Routine*). Esta divisão permite reduzir para o mínimo a latência da interrupção. No entanto, se se trata de uma interrupção que necessita apenas de um pequeno processamento este pode ser efetuado completamente na ISR. A nível de prioridades, uma ISR tem prioridade absoluta sobre as DSRs assim como uma DSR tem prioridade absoluta sobre as *threads*.

Nas ISRs são executadas as instruções com as interrupções e o escalonador desligados, ora quando acontecem, o processador somente se preocupa por processar a ISR. Neste caso como o escalonador está desligado, nenhuma API de sincronização do *kernel* pode ser utilizada pois estas efetuam interações com o escalonador. As DSRs são utilizadas quando o processamento da interrupção é mais complexo: neste caso são executadas, imediatamente após a ISR ter finalizado a execução. Estas DSRs são executadas com as interrupções e o escalonador ativo, o que pode levar com que a DSR não seja completada de uma forma contínua. Como as interrupções estão ativas, uma interrupção de maior grau de prioridade pode ocorrer, interrompendo a execução da DSR e lançar a respetiva ISR. Como a DSR é executada com o escalonador ativo, esta pode utilizar mecanismos de sincronização como por exemplo *semaphore* para sinalizar a uma *thread* que uma interrupção ocorreu. Mas esta particularidade também pode ser um inconveniente, pois caso uma *thread* tenha bloqueado o escalonador com a função "cyg_sched_lock" antes de ocorrer a interrupção, a DSR só será executada após a *thread* ter libertado o escalonador com a função "cyg_sched_unlock".

Para exemplificar a gestão de uma interrupção, a Figura 21 vem ilustrar o seu funcionamento, seguido da sua descrição passo a passo.

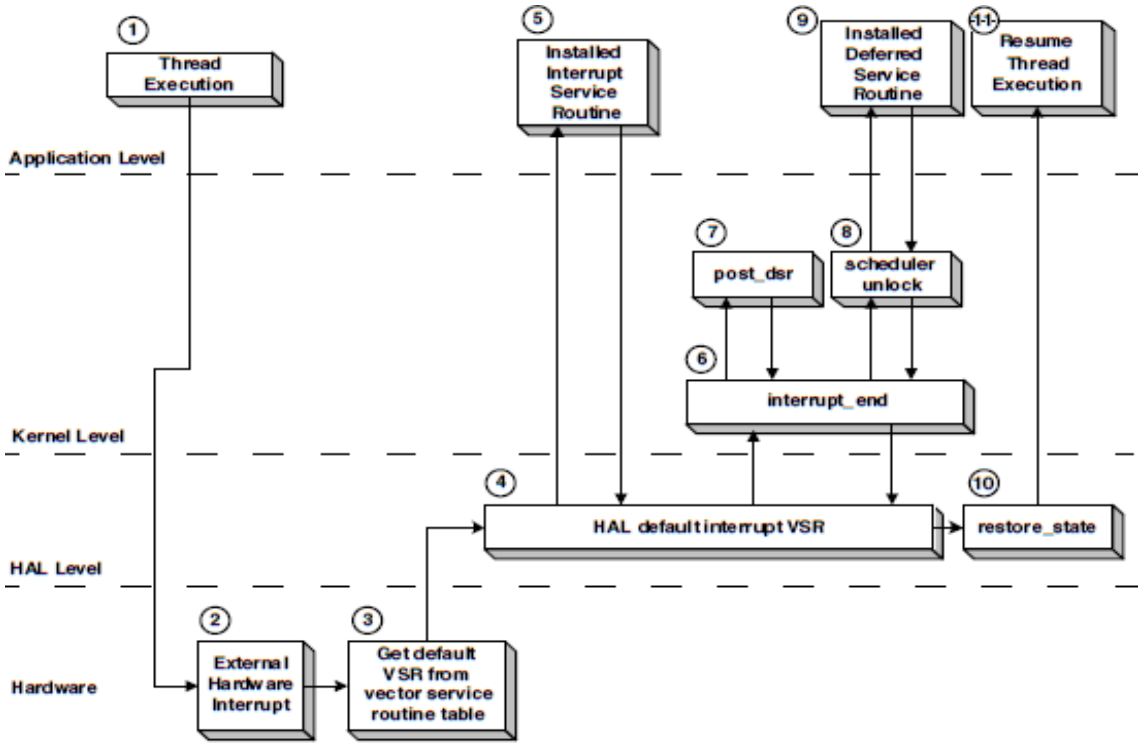


Figura 19: Manipulação de uma interrupção no eCos™ [12].

1. Execução de uma *thread*
2. Ocorrência de uma interrupção externa.
3. O processador obtém o apontador por defeito das interrupções a partir da tabela VSR.
4. É guardado o estado do processador, bloqueado o escalonador e as interrupções. Depois o VSR por defeito aponta para a localização das ISR e no conjunto das ISRs, é determinada qual deve ser executada após a função "hal_intr_decode" devolver o seu valor.
5. A ISR é executada ao nível da aplicação, e notifica ao *kernel* que a DSR tem que ser executada após esta terminar. A DSR só será executada caso não exista nenhuma DSR mais prioritária por executar.
6. Após a ISR estar terminada a função a função "interrupt_end" é chamada.
7. Se é necessário executar uma DSR, a função "interrup_end" chama a função "post_dsr".
8. Após "post_dsr" retornar o escalonador é desbloqueado.
9. Se o escalonador não for bloqueado por uma *thread* a DSR é executada (se existir), senão é executada a *thread*, neste caso está uma DSR ativa e o escalonador não está bloqueado.
10. Assim que a DSR acaba, a rotina "restore_state" é lançada.
11. Finalmente a *thread* é retomada.

3.2 O Processador RISC – OR1200

O processador OR1200, Figura 20, é um processador RISC de 32-bit com cinco estágios de *pipeline*. Possui suporte de memória virtual através das MMUs, cache de instruções e dados *direct-mapped*, módulos DSP (*Digital Signal Processing*), temporizadores/contadores, uma unidade de *debug,* uma unidade de controlo de interrupções e outra de gestão de energia, que são todas opcionais. Para finalizar, possui também um interface ao barramento Wishbone para memória de dados e de instruções. O *Instruction Set Architecture* (ISA) do OR1200 é composto por duas subclasses, o ORBIS32 (Anexo III) e o ORFPX32.

O ORBIS32 refere-se às instruções de *load/store*, as instruções de salto condicional e incondicional e as instruções para as operações sobre inteiros de 32-bits. O ORFPX32 que é opcional refere-se às instruções com inteiros de vírgula flutuante.

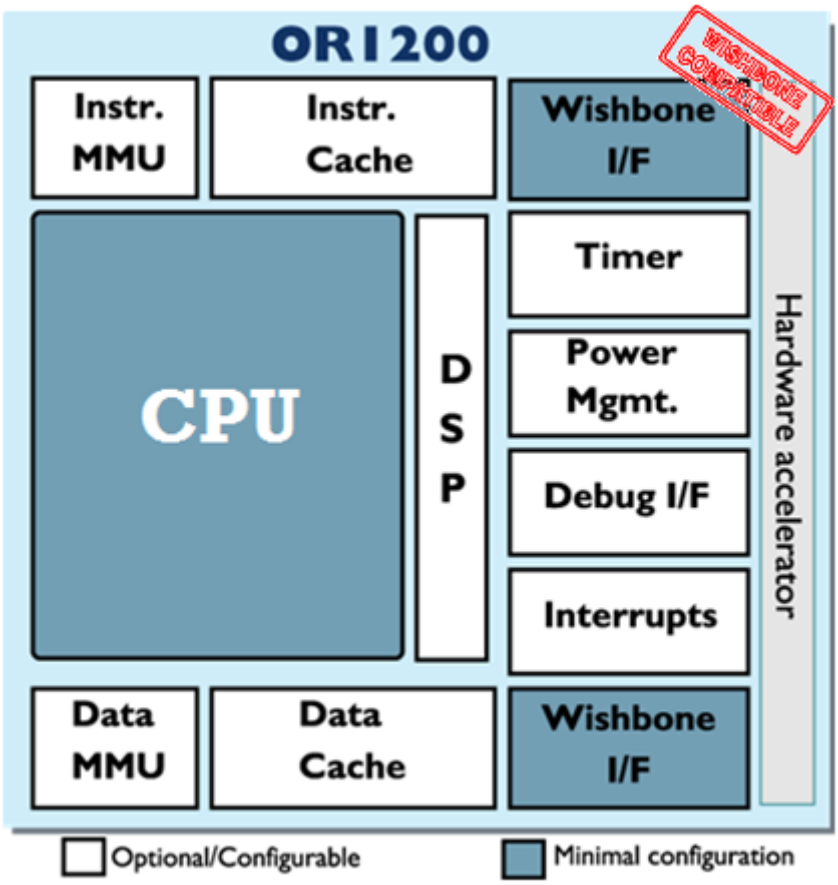


Figura 20: Diagrama de blocos - constituição do Processador OR1200 [13]

3.2.1 Unidade Central de Processamento - CPU

A unidade central de processamento, como o seu nome indica é o centro de todas as operações do processador, nela estão inseridas os cinco estágios de *pipeline*: *Fetch, Decode, Execute, Memory Access* e *Write Back*. O CPU é composta por quinze módulos diferentes que efetuam as suas próprias operações, como pode ser visto na Figura 21.

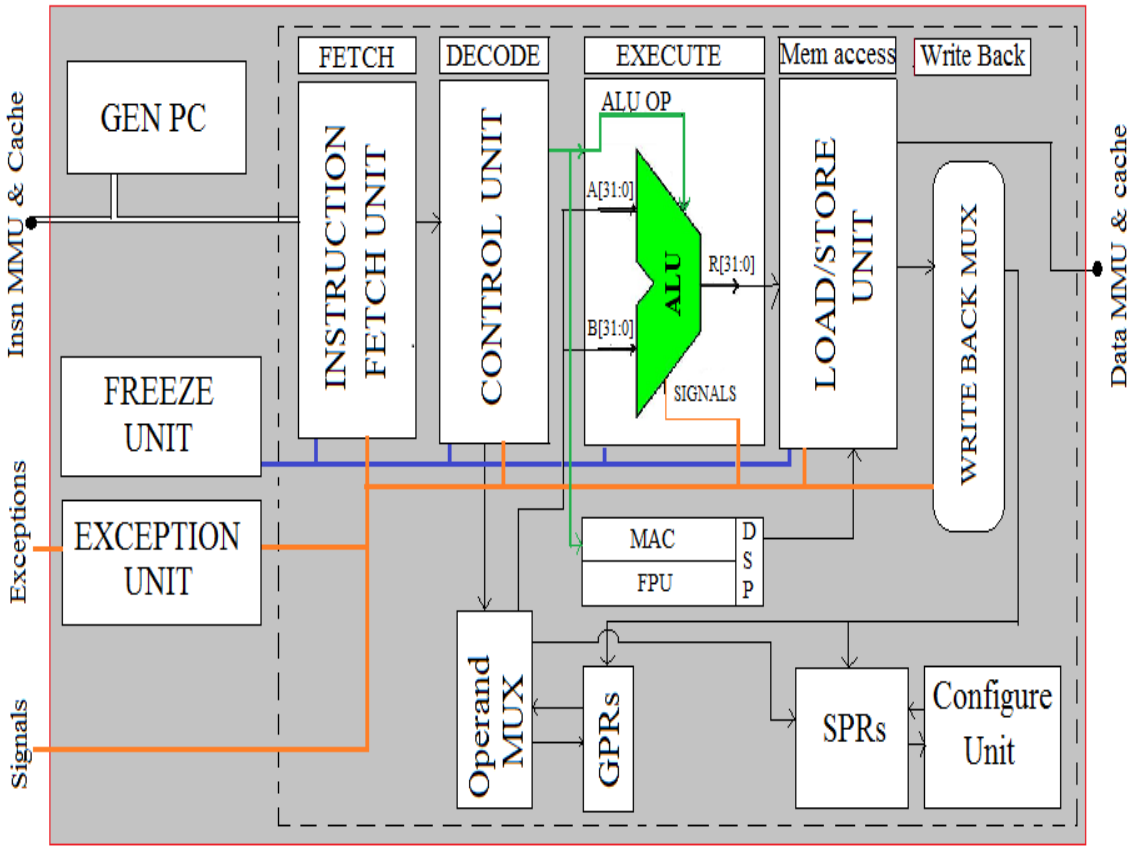


Figura 21: Diagrama de Blocos - constituição interna do CPU do OR1200.

A unidade de *fetch* das instruções é responsável por receber a instrução da cache de instruções e guardá-la para depois fornecê-la à unidade de controlo. É de salientar que cada vez que a nova instrução é guardada na unidade de *fetch,* a unidade *Gen PC* já gerou o novo valor do *Program Counter* que fornecerá sempre o endereço certo da próxima instrução. A unidade de controlo efetua a descodificação da instrução, dependendo desta, a sua saída pode fornecer à unidade *Operand*

Mux os registos de propósito geral (GPR) que serão usados, devolvendo os valores respetivo á unidade responsável por efetuar a execução da instrução.

A Unidade Lógica Aritmética (ALU) é responsável por efetuar as operações logicas e aritméticas, recebendo os operando 'A' e 'B', com o valor dos registos e/ou o valores imediatos e efetua a operação certa dependendo do valor devolvido por 'ALU_OP'. A sua saída devolve o resultado da operação e também caso seja necessário sinais de controlo como *Carry* e *Flag* entre outros. LOAD/STORE é a unidade responsável pelos acessos à memória, efetuando leituras ou escritas a memória de dados. A unidade "Write Back MUX" é composta por *multiplexers* e é responsável por devolver aos registos os valores após operação. A MAC *unit* (*Multiply And aCcumulate*), em conjunto com a FPU (*Floating Point Unit*) são DSPs. Esta primeira é responsável por efetuar operações de multiplicação e em segundo lugar por efetuar operações com vírgula flutuante. SPRs (*Special Purpose Register*) em conjunto com o módulo *Configure Unit* são responsáveis por efetuar leituras e escritas nos registos especiais de controlo e configuração do processador, assim como no registo de supervisão. Um exemplo da sua função é a responsabilidade que este tem por escrever no registo de controlo das interrupções. A *Freeze Unit* é uma unidade que serve para efetuar uma paragem no *pipeline* do processador, denominado *stall*, isto pode ser devido a um *hazard* ou simplesmente, uma espera por parte da unidade de *load/store* até que receba o dado correto. Esta unidade pode parar também a unidade *GenPC* de gerar o novo valor do *Program Counter*. Por fim, a unidade de exceções é responsável por identificar a ocorrência destas e efetuar a gestão necessária.

3.2.2 *Memory Managment Unit* e caches

O processador OR1200 vem com MMUs para memória de instrução e outra para memória de dados. Estas unidades são responsáveis por implementar a gestão da memória virtual. Essa gestão passa por traduzir endereços efetivos para endereços físicos que são realizados quando: há um pedido efetuado pela unidade de *fetch* no caso das instruções, ou um pedido efetuado pela unidade de LOAD/STORE no caso dos dados. As *page table* são estruturas de dados usados para guardar o mapeamento entre o endereço virtual e o endereço físico respetivo. Na Figura 22, está representada de uma forma simples como a MMU traduz um endereço efetivo para físico.

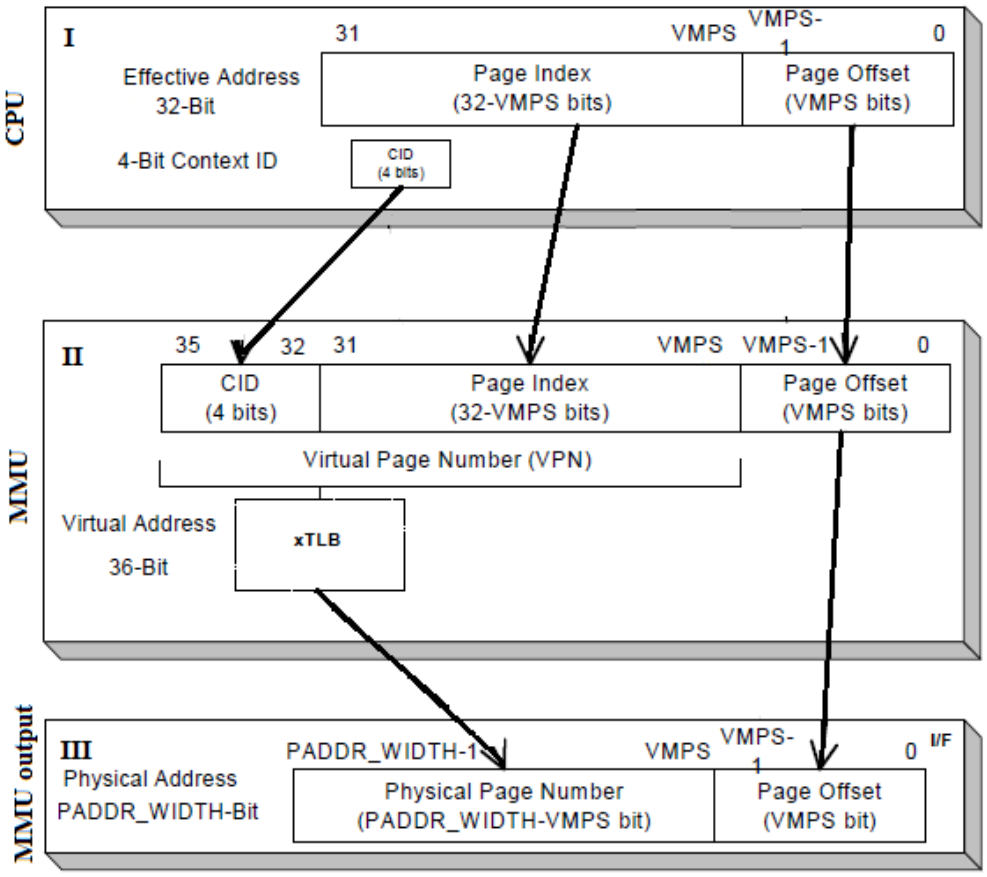


Figura 22: Diagrama de blocos MMU – Tradução do endereço efetivo para o endereço físico [14]

Em primeiro lugar o CPU fornece o endereço efetivo e 4 bits de contextualização, depois a MMU converte estas duas parcelas no endereço da página na memória virtual. Finalmente recorrendo à TLB (*Translation Lookaside taBle*) é determinado qual o endereço físico.

As MMUs também fornecem mecanismos de proteção ao acesso às *page tables*, por exemplo, caso a unidade de *fetch* ou *load/store*, queiram aceder a uma página onde os dados estão protegidos, uma exceção do tipo "Page Fault" será criada. Existe outro tipo de exceção que pode ser gerado pelas MMUs, este é uma exceção do tipo "TLB Miss" que é quando o endereço efetivo não corresponde a nenhuma entrada da tabela. Resumindo, a MMU em conjunto com a unidade de gestão de exceções fornece o suporte necessário para o RTOS implementar um ambiente de memória virtual paginada com proteção no acesso a essas áreas.

As caches são unidades que armazenam dados de forma transparente, de modo a que um pedido futuro a esses dados seja devolvido ao CPU de uma forma muito mais rápida. Esta rapidez deve-se

a quando é feito um pedido a cache e se esta tiver o dado (cache hit), o CPU só precisa de ler o dado que está guardado na cache. Caso contrário (cache miss) o CPU tem que efetuar o pedido a memória externa que é mais lenta. Deste modo, quanto mais solicitações puderem ser servidas pelas caches, mais rápido será o desempenho geral do sistema.

A cache de instruções e a cache de dados no OR1200 estão estritamente ligadas às MMUs correspondentes. Ambas são *direct-mapped*, isto é, para uma certa posição da memória principal esta só poderá corresponder numa entrada da cache, pois estas não possuem uma política de substituição que lhes permita associar várias posições da cache para a mesma posição da memória principal, como acontece nas caches *'N'-way associative* ou *fully-associative.*

3.2.3 O barramento WISHBONE e sua interface

O barramento Wishbone [15] é um barramento de dados destinado a efetuar uma ligação estandardizada entre diferentes IP *cores*, podendo nesta arquitetura efetuar ligações entre o OR1200, memória externa, módulos UART e outros periféricos modulares. Para comunicar com este barramento, o OR1200 possui dois módulos que efetuam a tradução do barramento de dados e de instruções internos, para o barramento *standard* Wishbone, denominados de "wb_biu".

No OR1200 o barramento Wishbone funciona como *multi-master* e *multi-salve*: existindo um barramento para dados e outro para instruções. A Figura 23 representa como é efetuada a ligação entre um módulo *slave* e o *master*.

Neste barramento existem catorze sinais diferentes, estes são:

- RST e CLK: são os sinais de *clock* e *reset* de 1-bit, fornecem ao barramento o sinal de relógio necessário para a transferência de dados, assim como o sinal de *reset* quando acontece um erro na transmissão dos dados. O sinal de *clock* é fornecido pela unidade que gera o *clock* para o processador e afins.
- ADR_O: fornece o endereço de saída, dado pelo *master* ao *slave*, este sinal tem 32-bit.
- DAT_I e DAT_O: são sinais de 32-bit (para o OR1200) para os dados de entrada e saída.
- WE: este sinal de 1-bit é fornecido pelo *master* a um *slave*, para informar que se trata de uma operação de escrita '1' ou leitura '0'.

- SEL_O: este sinal 4-bit é fornecido pelo *master* para informar quais os *bytes* na *word* serão usados para troca, isto é, num dado de 32-bit SEL_O define se são para ler os 4 *bytes* (SEL_0= 0b1111) ou por exemplo, somente o primeiro (SEL_O= 0b0001).
- STB_O e STB_I: este sinal de 1-bit indica que está a decorrer uma transferência válida, no caso do *slave* quer dizer que ele foi selecionado e só ele responderá pelo barramento.
- ACK_I: este sinal fornece ao *master* a confirmação que um dado foi bem recebido ou enviado e que o ciclo terminou.

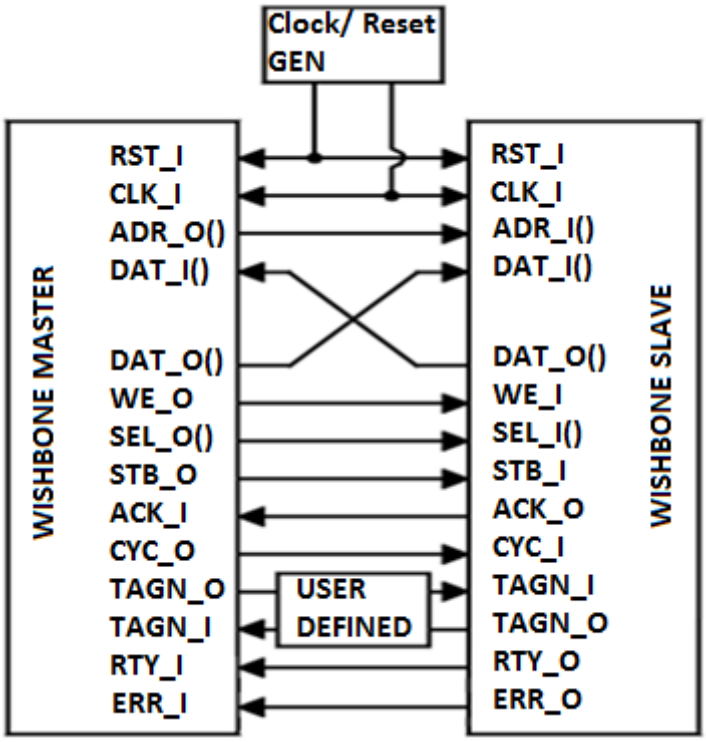


Figura 23- Esquema de ligação a barramento Wishbone [15]

- CYC_O: este sinal informa que existe um ciclo iniciado e que está em progresso.
- RTY_I: *retry é* um sinal enviado pelo *slave* ao *master* para pedir o recomeço do ciclo, confirmando que o ultimo ciclo não finalizou corretamente.
- ERR_I: o sinal de erro é enviado pelo *slave* para indicar ao *master* que um erro surgiu e que o ciclo não se completará.
- TAGN_I e TAGN_O: são sinais adicionais que fornecem informações para módulos especiais (*cache control*, interrupções etc.)

Para entender como é efetuado o ciclo de leitura e escrita através do barramento, recorrendo a Figura 24 que representa uma operação de leitura, seguida de uma operação de escrita, operada pelo *master,* a uma memória externa.

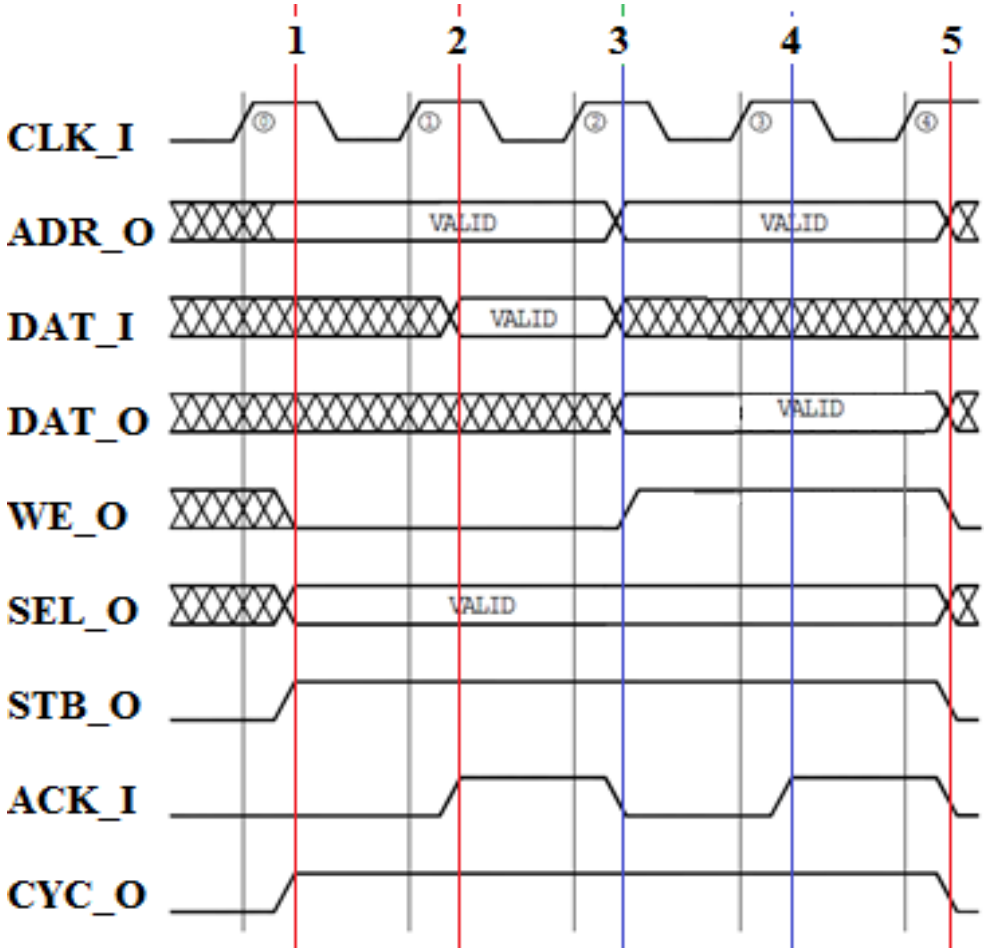


Figura 24*:* Wishbone *Master Signal-* ciclo de leitura e escrita

Em '1' o *master* inicia o ciclo, colocando os sinais CYC_O e STB_O a um, em simultâneo, identifica o número de bytes (SEL_O) a serem usados, a operação (leitura por WE_O está a '0') e o endereço de leitura. Não especifica DAT_O por se tratar de uma operação de leitura. Em '2', o *master* recebe o *acknowledge* por parte do *slave* e o dado já se encontra disponível em DAT_I. Em '3' o master efetua uma operação de escrita, WE_O passa a um, assim como é enviado o novo endereço e o dado a partir de DAT_O. Em '4', o *slave* devolve um *acknowledge*, informando ao *master* que recebeu o dado. Em '5' o *master* liberta o barramento ao recolocar CYC_O e STB_O a zero.

3.2.4 Os restantes periféricos opcionais

Está secção descreve os restante periféricos opcionais mencionados no subcapítulo 3.2 e representados na Figura 20, estes são o módulo do temporizador/contador, a unidade de *debug*, a unidade de gestão de energia e o controlador de interrupções. Estes foram reagrupados numa única secção, por se tratar de módulos que não influenciaram diretamente a modelação e implementação deste projeto.

3.2.4.1 Temporizador/contador

Este módulo temporizador/contador, que obtém o seu sinal de relógio através do sistema, é utilizado pelo RTOS para medir com precisão o tempo e o escalonamento das tarefas. A sua saída gera uma interrupção que poder ser mascarada. É composto por dois registos de controlo TTMR e TTCR. TTMR (*Tick Timer Master Register*) permite ativar/desativar o temporizador/contador e especificar em que modo funcionará. TTCR (*Tick Timer Counter Register*) é o registo onde é incrementado/decrementado o temporizador/contador. Este módulo permite três modos de operação diferente; *Auto-restart timer, One-shot timer* ou *Continuous timer*. A Figura 25 representa como é composto este módulo temporizador/contador.

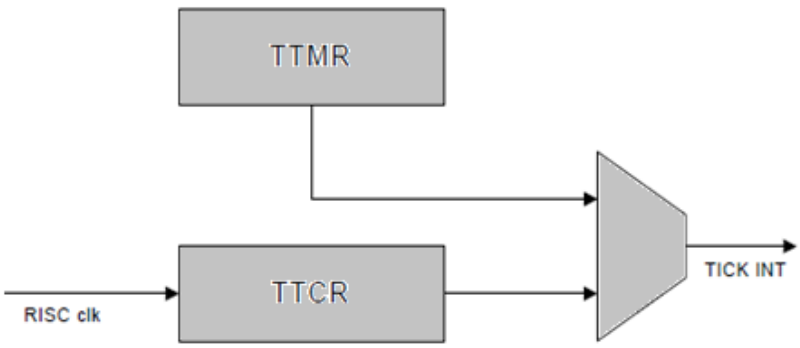


Figura 25: Diagrama de Blocos- Temporizador/Contador [14]

3.2.4.2 Unidade de *Debug*

A unidade de *debug* permite efetuar o teste e a validação do *software*, pois fornece ferramentas como *breakpoint, watchpoint* assim como durante a execução do programa, fornece também o

valor do *Program Counter* e dos registos (GPRs e SPRs). O par DVR/DCR, Figura 26, é usado para comparar o endereço efetivo da instrução no momento do *fetch* (IF EA) ou da unidade de LOAD/STORE (LS EA), com o dado obtido pela memória de dados (LS data), esta comparação que ainda pode ser mais complexa cria durante a execução do programa, dependendo do registo DMR (*Debug Mode Register*), os *watchpoint* e *breackpoint* previamente configurados.

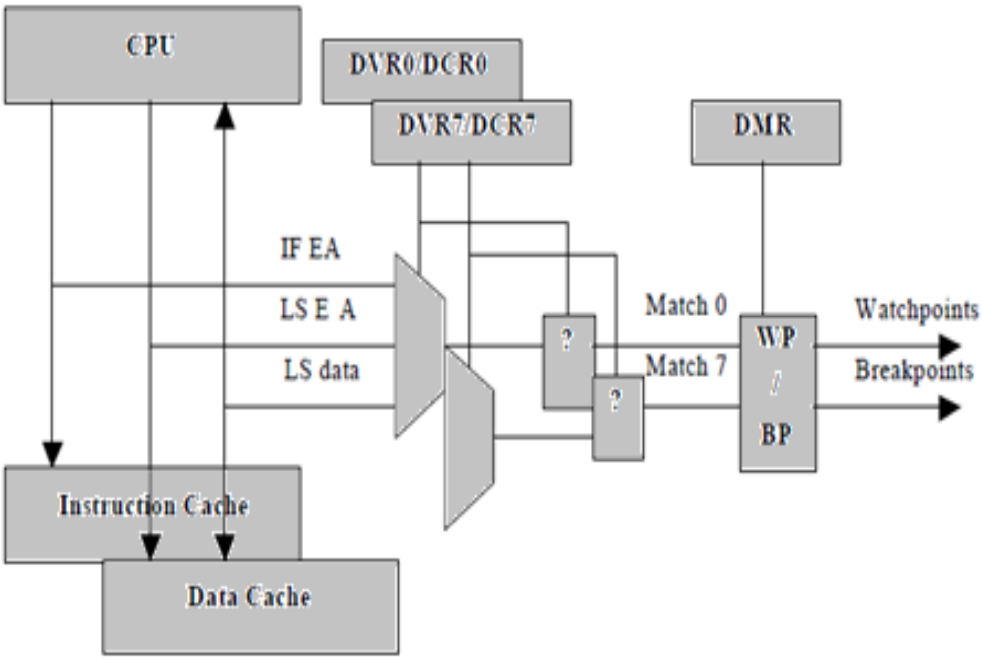


Figura 26: Diagrama de blocos- Unidade de *Debug* [14]

3.2.4.3 Unidade de gestão de energia

A unidade de gestão de energia, *power management,* permite configurar o processador para diversos modos de modo a minimizar o consumo de energia. Existem três modos: *slow down feature, doze mode e sleep mode.*

O modo *slow down feature,* é controlado por *software* com a configuração de 4-bits do registo PMR. Um valor mais baixo neste registo (0x0) coincide com um maior desempenho por parte do processador. Sendo que o RTOS controla o desempenho aumentando ou diminuindo o valor do registo PMR, sendo que quanto maior for o valor a frequência do clock diminui, diminuindo também o consumo de enrgia. O modo *doze* quando ativo, suspende o decorrer das instruções por

parte do CPU, pois desabilita o *clock* do CPU, deixando apenas ativo o temporizador/contador e o PIC. Quando surge uma interrupção o *clock* do CPU é reativado e o processador deixa de estar no modo *doze.* No último modo, *sleep mode,* todas as unidades internas são desabilitadas, pois o sinal de relógio não é fornecido e só voltará ao modo normal quando ocorrer uma interrupção.

3.2.4.4 Programable Interrupt Controller

O OpenRISC 1200 possui um controlador de interrupções programável com 32 pinos de entrada, sendo que o pino '0' e '1' estão sempre habilitados e conectados às prioridades altas e baixa respetivamente. Os restantes 30 pinos são *maskable*, isto é, estas interrupções podem ser ignoradas pelo sistema se no respetivo registo estiver '0'. O PIC também é composto por três registos de propósito especial; PICMR, PICSR e PIPR, ilustrados na Figura 27 permitindo que as interrupções sejam ativas por estado (0 ou 1) e/ou por transição de estado (0 para 1).

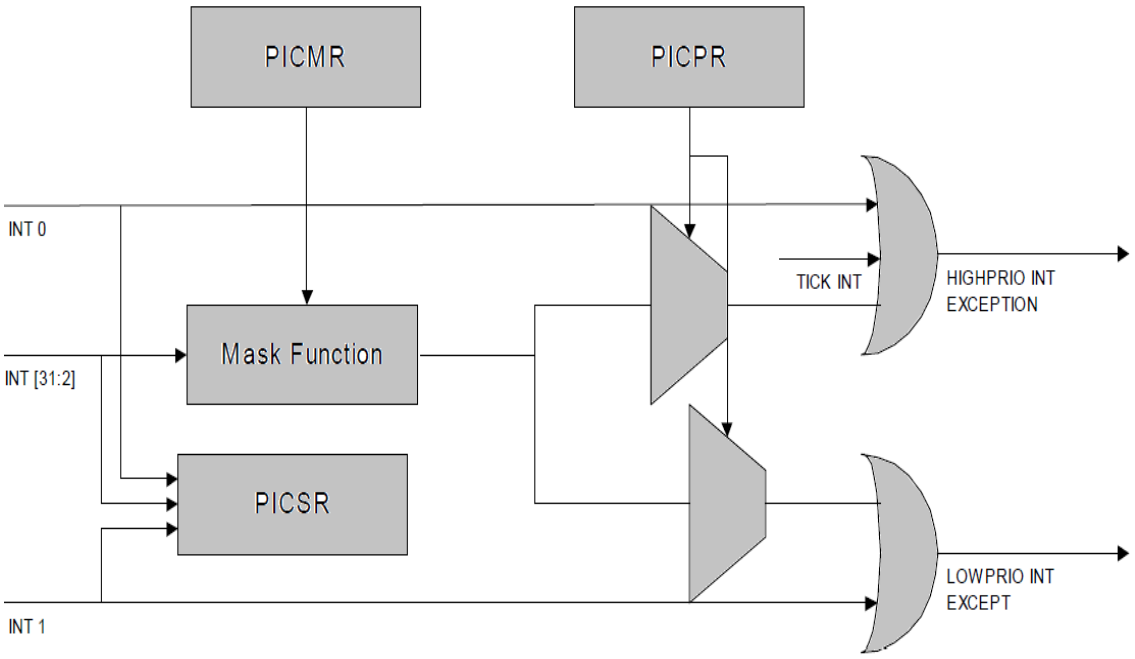


Figura 27: Diagrama de blocos- *Programmable Interrupt Controller (PIC)* [14]

O registo de controlo de interrupções mascaradas (PICMR) é um registo de 32 bits que funciona como supervisor e é usado para mascarar ou desmascarar as interrupções. Por exemplo, caso o valor do registo seja '0x0', isto quer dizer que todas as interrupções estão mascaradas. O registo PICSR é o registo do estado das interrupções, ou seja, serve unicamente para verificar se as

interrupções estão ativas ou desativas. As interrupções não podem ser 'limpas' através deste registo, são feitas por *hardware*, mas, caso se trate de uma ativação por transição de nível numa interrupção, esta terá que ser 'limpa' ao escrever um '1' no bit respetivo do registo, isto para certificar que a *latch* venha ao valor '0'. O registo PICPR é o registo de controlo de prioridade e serve para definir se a prioridade é alta ou baixa. Estes registos PICMR, PICSR e PICPR são acedidos com as instruções 'l.mtprs' e 'l.mfspr' presentes no ISA do OR1200.

3.3 ORPSoC – *OpenRisc reference Platform System-on-Chip*

O ORPSoC [16] é um projeto desenvolvido pela comunidade OpenCores, trata-se de um System-on-Chip que é constituído pelo processador OR1200 ao qual foram acoplados de uma forma *loosely-coupled,* através do barramento Wishbone, diversos periféricos adicionais.

Neste subcapítulo são referidos os módulos ligados ao processador através do barramento Wishbone que são: a memória externa, o módulo UART e os *arbiters* que controlam a troca de informação pelo barramento de dados e instrução.

3.3.1 Os *arbiters* do barramento de dados e de instruções

Os *arbiters* são um componente fulcral quando é utilizado o barramento, pois estes são responsáveis pelo controlo do seu acesso, isto é, com base na prioridade do *master* determinam qual dos *masters* e qual dos *slaves* estão a comunicar pelo barramento. Existem três *arbiters* diferentes o *arbiter* para instruções 32-bits, para dados de 32-bits e o *arbiter* de acesso ao *byte*, para módulos UART ou GPI/O.

Observando a Figura 28, vê-se para o barramento de instruções que este possui um *master,* o CPU e um *slave* a memória externa. Para o barramento de dados, o *arbiter Dbus* faz a ligação como dois *slaves,* a memória externa e o *arbiter ByteBus.* No primeiro caso, é quando o CPU quer escrever ou receber um dado da memória. O segundo caso, quando passa pelo arbiter *bytebus*, trata-se de uma comunicação especial para periféricos como a UART, que enviam dados com tamanhos inferior a 32-bit. Estes enviam ou recebem os dados *byte* a *byte,* logo o dado devolvido pelo *arbiter Dbus* é dividido em 4 *bytes* e só é passado 1 *byte* de cada vez.

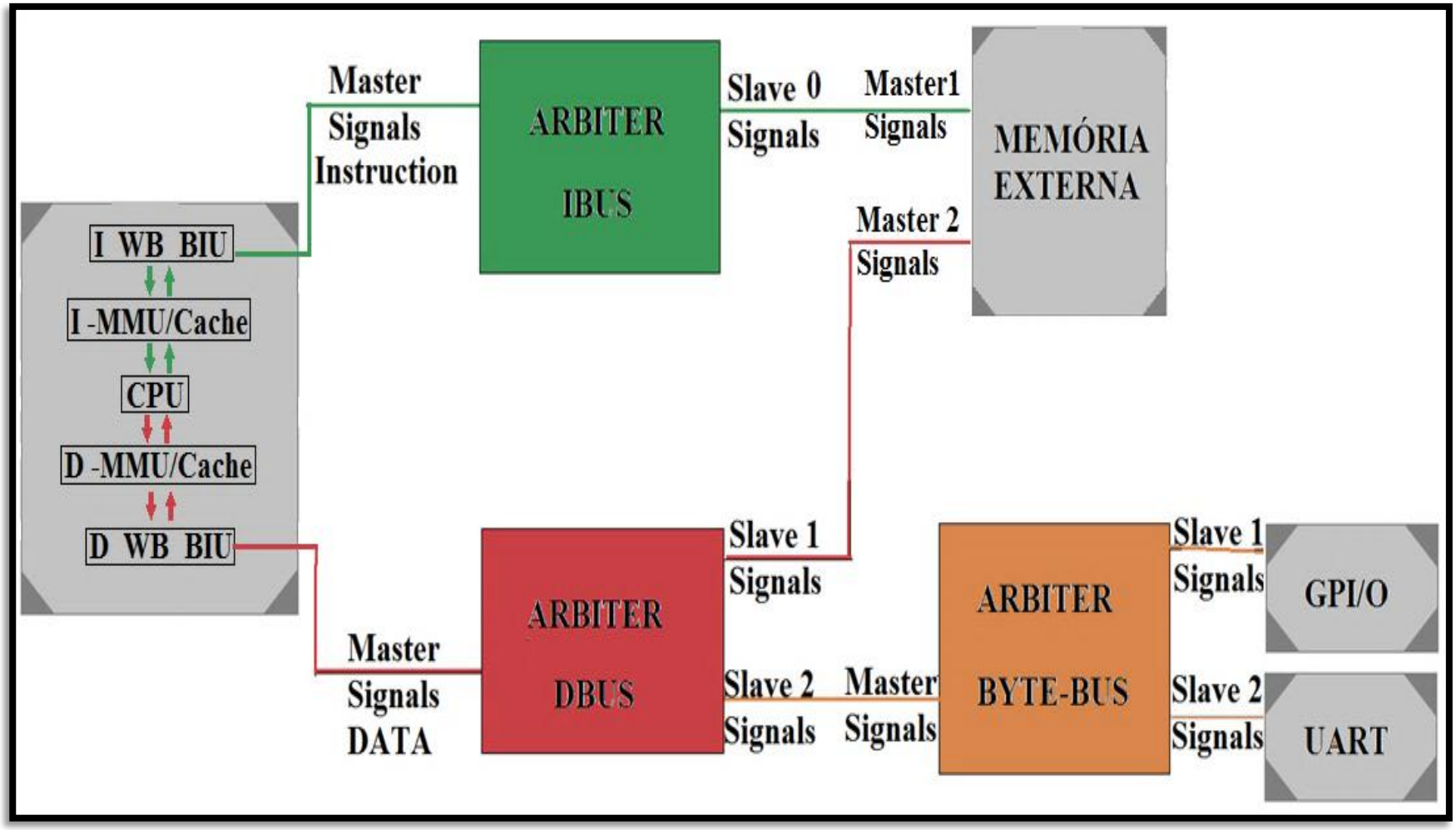


Figura 28: Diagrama de blocos - Ligações dos *arbiters* entre *slave* e *master*

3.3.2 Memória Externa

A memória externa é a mesma para memória de código assim como para a memória de dados. Esta terá que ter um tamanho mínimo de 65 Kbytes para poder guardar a aplicação com as bibliotecas do eCos™. Esta é *dual-port* com acesso síncrono, isto é, permite acesso a escrita e leitura em simultâneo, como se pode ver na Figura 29. Para efetuar comunicações através do barramento Wishbone possui um módulo de compatibilidade com este barramento, sendo que o primeiro *master* a pedir o barramento (instrução ou dados) será o primeiro servido.

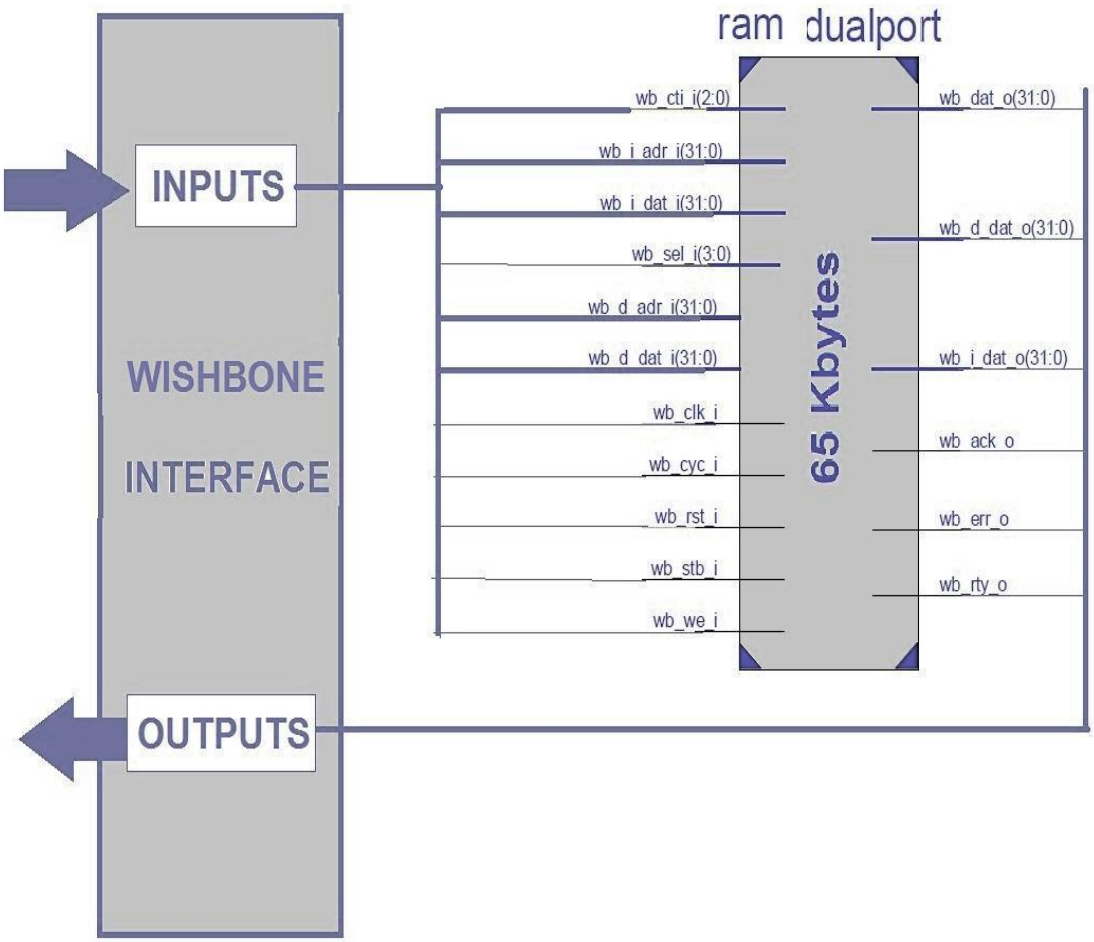


Figura 29: Diagrama de blocos- memória externa de dados e de instruções

3.3.3 O módulo UART 16550

O módulo de porta série UART16550 é um IP *core* desenvolvido pela comunidade OpenCores e integrado no projeto ORPSoC para acrescentar mais um periférico ao OR1200, visto que possui compatibilidade com o barramento Wishbone. Este módulo possui suporte para interrupções, registos de controlo para configurar interrupções, *buffer* de receção de dados e *buffer* de transmissão de dados, como se pode observar na Figura 30. Por defeito este módulo comunica com um *baudrate* de 11500 bps (bit por segundo), 8 bit de dados, 1 *start* bit e 1 *stop* bit.

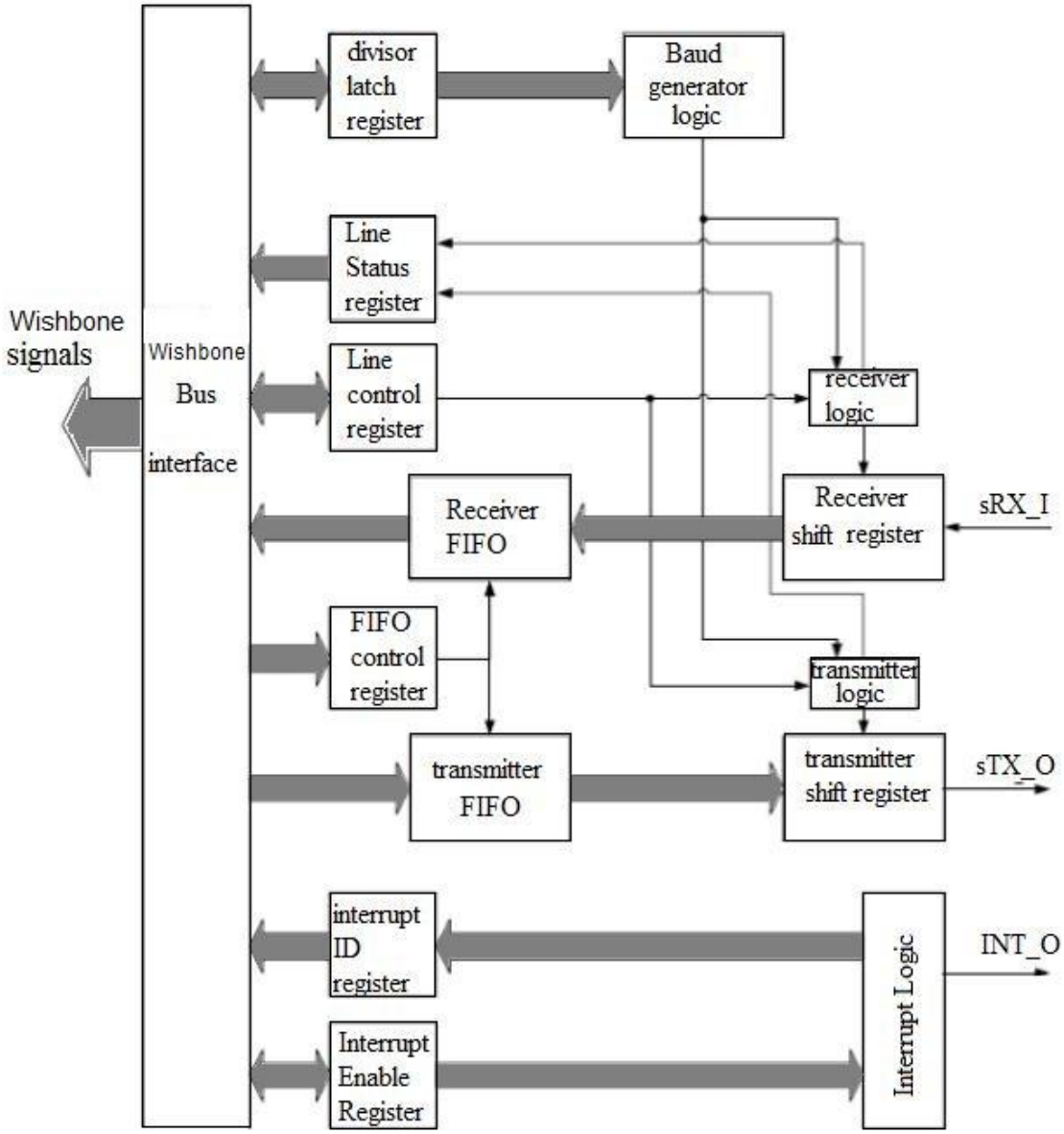


Figura 30: Diagrama de blocos- módulo UART do OR1200 [**17**]

Capitulo 4

Modelação e Implementação

Este capítulo apresenta o projeto das migrações a realizar e as técnicas usadas para a implementação do projeto. Assim, irão ser descritos os novos módulos de *hardware* adicionados e como estes estão ligados ao processador, assim como as alterações necessárias ao *software* para que as funcionalidades dos novos módulos possam ser usadas por este. No final deste capítulo é apresentado o esquema físico da implementação efetuada.

4.1 Migração de funções do RTOS para *hardware*

A migração de funções de um RTOS comercial para *hardware* possui algumas limitações de implementação. Para conseguir ter um RTOS híbrido, com funções em *software* e outras em *hardware*, é necessário manter a estrutura funcional do RTOS, pois caso contrário a coordenação/coerência entre funções operadas por *software* e funções operadas por *hardware*, não é verificada.

Sendo assim para efetuar a migração de uma função em *software* para *hardware* com sucesso é necessário ter em mente três conceitos de implementação das funções:

- Uma função em *software* que recebe argumentos, deverá, quando implementada em *hardware*, passa-los através da instrução ou caso seja mais simples, o módulo de *hardware* deverá saber a sua localização para poder opera-los.
- Caso uma função 'A' em *software* efetue a chamada de outra função 'B', o módulo em *hardware* responsável por efetuar essa operação terá que chamar a função 'B' no devido momento, caso esta não esteja implementada em *hardware*. Caso a função 'B' também esteja implementada em *hardware*, o módulo de *hardware* da função A, ativará o módulo 'B' em *hardware*.
- Qualquer devolução de argumentos por parte de uma função em *software* terá que ser cumprido se esta for migrada para *hardware*, assim como o *output* originado durante a operação da função, terá que corresponder ao *output* originado pela função em *software*.

Agora que foram especificados os três conceitos para migrar funções para *hardware*, os subcapítulos seguintes especificam quais foram os pré-requisitos a nível do *software* seguido da modelação do RTOS em *hardware* – Hrtos.

4.1.1 Pré-requisitos de *Software*

Para que a migração seja compatível com o resto do RTOS, algumas operações importantes têm que ser efetuadas. A primeira é o acréscimo de novas instruções ao *assembler*, para este traduzir instruções para código máquina, pertencentes ao módulo em *hardware*. A segunda passa por alterar as funções a migrar nas bibliotecas do eCos™, usando o *assembler inline* caso seja necessário.

4.1.1.1 Adição de novas instruções

As novas instruções à acrescentar ao ISA do processador estão apresentadas na Tabela 5. Estas instruções possuem um formato do tipo "le.X imm;" em que a mnemónica 'le'[2] refere-se a uma instrução para o *hardware* RTOS, a mnemónica representada com 'X' refere-se a classe onde a

[2] As instruções que comecem por 'l.' são instruções do ORBIS32 'lf.' são instruções para unidade de operações com virgula flutuante, sendo assim para manter a mesma coasão 'le.' Representa operações para o RTOS em hardware, em que o 'e' refere-se a eCos™.

função implementada em *hardware* existia como método. Por exemplo 'ht' refere-se a uma instrução que é método da classe "hardware_Thread". Para finalizar o 'imediato' (Imm) refere-se a um valor inteiro que identifica qual é a função que se está a chamar.

Tabela 5: Instruções adicionadas ao ISA OR1200

Instrução - Bit	31-26	25-21	20-16	15-0	Referência/classe
le.ht imm,	6b111111	5b00001	0	Imm	*Hardware thread*
le.scht imm,	6b111111	5b00010	0	Imm	*Scheduler Thread*
le.t imm,	6b111111	5b00011	0	Imm	*Thread*
le.tqueue imm,	6b111111	5b00100	0	Imm	*Thread queue*
le.sched imm,	6b111111	5b00101	0	Imm	*Scheduler*

Estas novas instruções serão acrescentadas ao *assembler*, mais concretamente no ficheiro 'or32-opc.c' da *toolchain*, que se encontra na diretoria /gnu-src/binutils-2.20.1/opcodes/or32-opc.c. Após efetuar esta alteração, esta só será concretizada quando a *toolchain* é compilada, criando deste modo, o *assembler* com as novas instruções e as restantes ferramentas para o OpenRISC.

4.1.1.2 Alteração das bibliotecas do eCos™

As alterações às bibliotecas do eCos™ são realizadas para acrescentar as novas instruções com a ajuda do *assembler inline*, assim como as *macros,* que definirão se a função será implementada em *software* ou *hardware*. Estas alterações estão exemplificadas na Figura 31.

A instrução "le.hrtos 0x1" é inserida no código recorrendo ao *assembler inline*. Esta ferramenta usada para otimizar o código, servirá para chamar uma instrução específica do processador, neste caso a instrução para realizar por *hardware* a função "schedule".

Como se pode observar a função "schedule" possui uma *macro* 'CYG_HRTOS_SCHED_BM_SCHEDULE'. Na hora de gerar o ficheiro objeto relativo à compilação da biblioteca, caso a *macro* esteja definida, implementará a função "schedule" do escalonador *bitmap* em *hardware*.

Caso contrário a função permanecerá implementada em *software*, originalmente disponibilizada na respetiva biblioteca do eCos™.

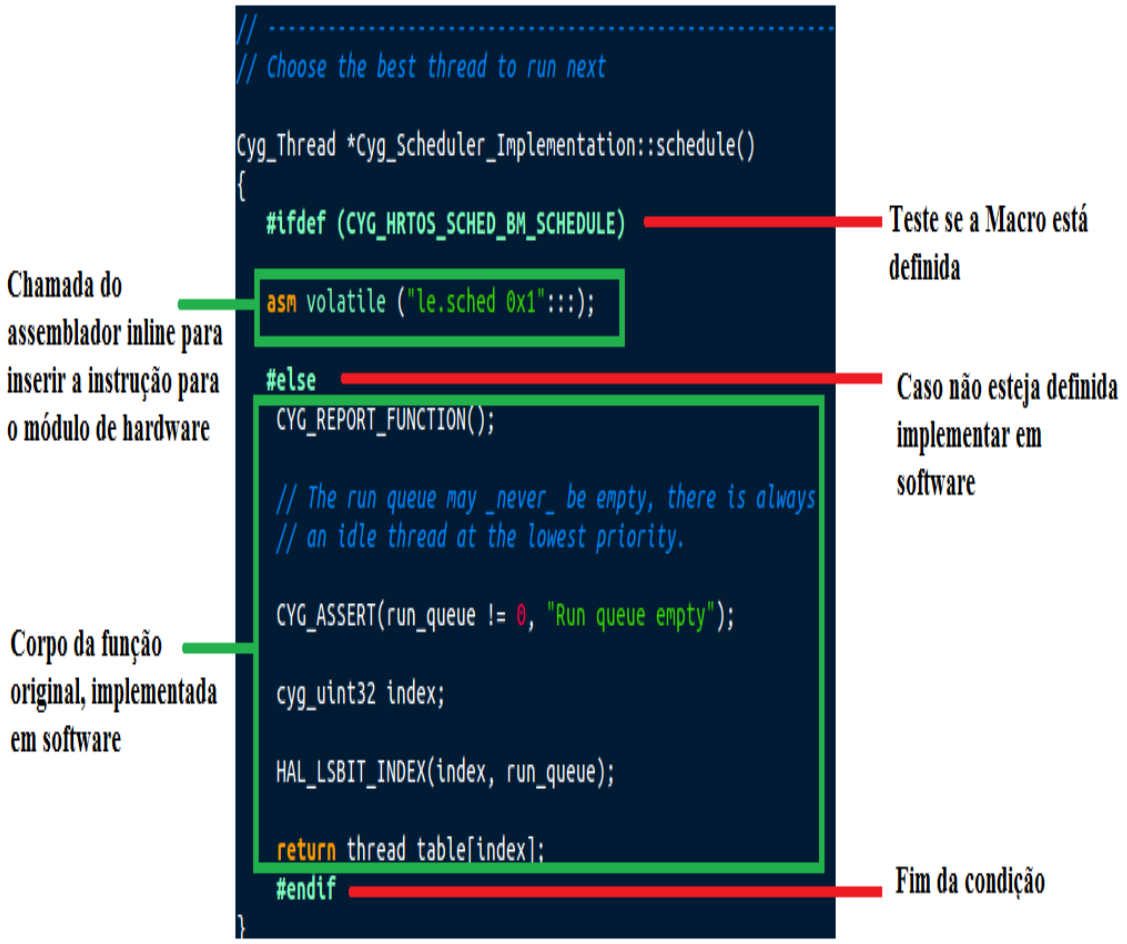

// Choose the best thread to run next
Cyg_Thread *Cyg_Scheduler_Implementation::schedule()
{
#ifdef (CYG_HRTOS_SCHED_BM_SCHEDULE)
asm volatile ("le.sched 0x1":::);
#else
CYG_REPORT_FUNCTION();
// The run queue may _never_ be empty, there is always
// an idle thread at the lowest priority.
CYG_ASSERT(run_queue != 0, "Run queue empty");
cyg_uint32 index;
HAL_LSBIT_INDEX(index, run_queue);
return thread table[index];
#endif
}
Teste se a Macro está definida
Chamada do assemblador inline para inserir a instrução para o módulo de hardware
Caso não esteja definida implementar em software
Corpo da função original, implementada em software
Fim da condição

Figura 31: Exemplificação da alteração das bibliotecas do eCos™

4.1.2 O módulo *hardware* RTOS – Hrtos

O módulo 'Hrtos', refere-se à implementação por *hardware* do RTOS. Nele estarão contidas todas as máquinas de estado e operações, que implementarão por *hardware* as funções migradas. Além das funções, este módulo terá que ter uma ligação *tightly-coupled* ao *pipeline* do processador e aos seus registos de propósito geral, assim como um módulo de compatibilidade Wishbone para poder aceder ao barramento de dados. Para além dessas ligações o Hrtos deverá possuir uma unidade própria de descodificação das instruções.

4.1.2.1 Funcionamento geral de Hrtos

O módulo Hrtos comportar-se-á como uma unidade de coprocessamento, no entanto esta nunca será executada em simultâneo com o processador visto que ambas partilharão os registos de propósito geral e o acesso a memória externa. Sendo assim é necessário criar um sinal que bloqueie o *pipeline* do processador quando este está a tentar processar uma instrução que não lhe é destinada. Este sinal é gerado pela unidade de descodificação das instruções interna ao Hrtos, ver secção 4.1.2.2.

A Figura 32 exemplifica como funcionará o Hrtos. Numa primeira fase o processador estará a executar instruções que passam pelo diversos estágios do seu pipeline. A verde a unidade de *decode* do CPU verificará que existe uma instrução ilegal e preparar-se-á para criar uma exceção. No entanto como a informação será passada ao Hrtos para este efetuar a descodificação das suas instruções, caso seja uma instrução para este, ativará o sinal "insn_rtos" que estará ligado a unidade de *freeze* do processador, efetuando-se uma paragem na execução do *pipeline.* Em simultâneo Hrtos ativará o *enable* à FSM (*Finite State Machine*) da função correspondente. Assim que a FSM termina Hrtos acabará de operar, será substituída a instrução ilegal por uma instrução '*NOP*' e a execução será devolvida ao processador.

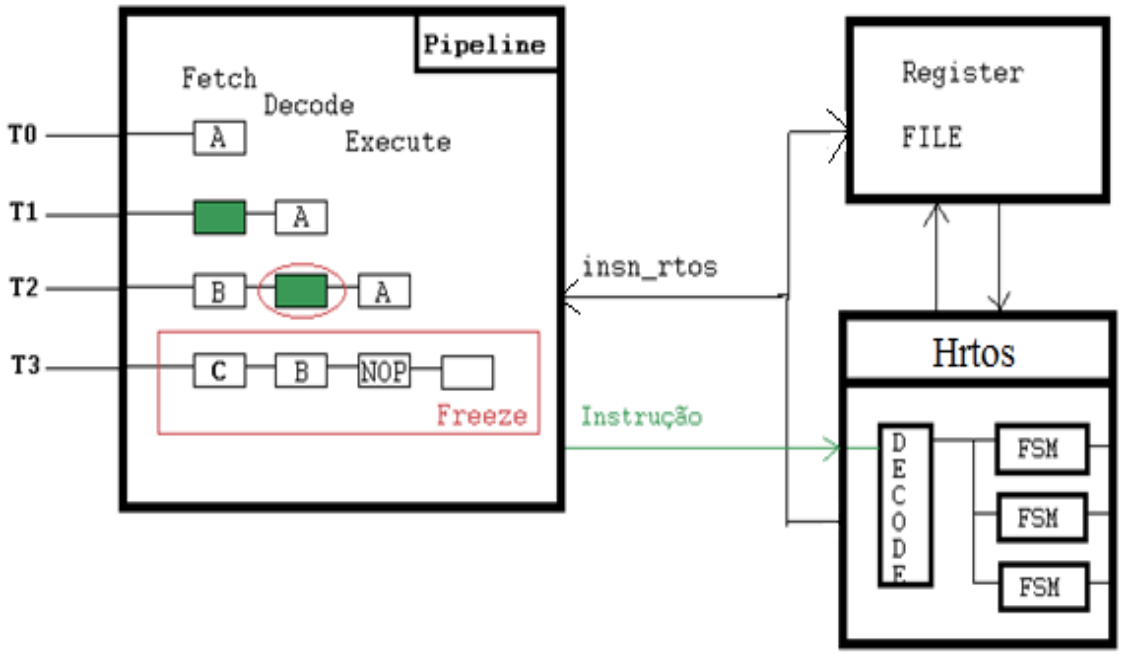


Figura 32: Funcionamento do Hrtos

4.1.2.2 A unidade de *decode* de Hrtos

A unidade de descodificação de instrução interna ao Hrtos permitir-lhe-á saber quando uma instrução deve ser processada por si ou pelo CPU. Esta unidade deverá ter as seguintes entradas:

- 'Hrtos_reset'- 1bit para efetuar *reset* e repor os valores de origem.
- 'Hrtos_clk' fornecerá o sinal de *clock* do relógio para as operações síncronas.
- 'Hrtos_fsm_finish'- 1bit que indicará que todas as FSM estão inativas/completadas
- 'Id_insn_i'- 1 *wire* de 32 bits que fornecerá a instrução que se encontra no 2°estagio do *pipeline* do processador.

E as seguintes saídas:

- 'Hrtos_d_insn'- *wire* de 32 bits que fornecerá a instrução a respetiva FSM.
- 'Hrtos_insn_e'- *wire* que indicará que Hrtos está a operar.
- 'Insn_rtos_reg' - registo que sinalizará a unidade de *freeze* para efetuar a paragem/reativação do CPU.

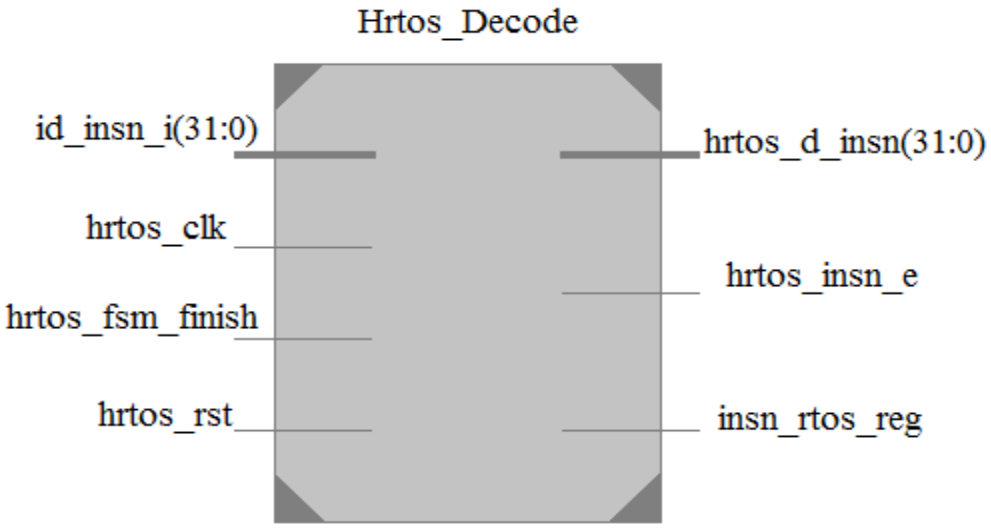


Figura 33: Esquema da unidade Hrtos_Decode

Caso o *opcode* recebido pelo sinal 'id_insn_i', dado pelo CPU, seja uma instrução para o Hrtos este ativará o sinal 'hrtos_insn_e' que por sua vez ativará a FSM adequada, recorrendo ao sinal 'hrtos_d_insn'. Para informar o CPU que o Hrtos começará a operar e tomar controlo dos registos

e da memória externa será usado o sinal 'insn_rtos_reg'. A Figura 34 representa o diagrama de estados do sinal 'insn_rtos_reg'.

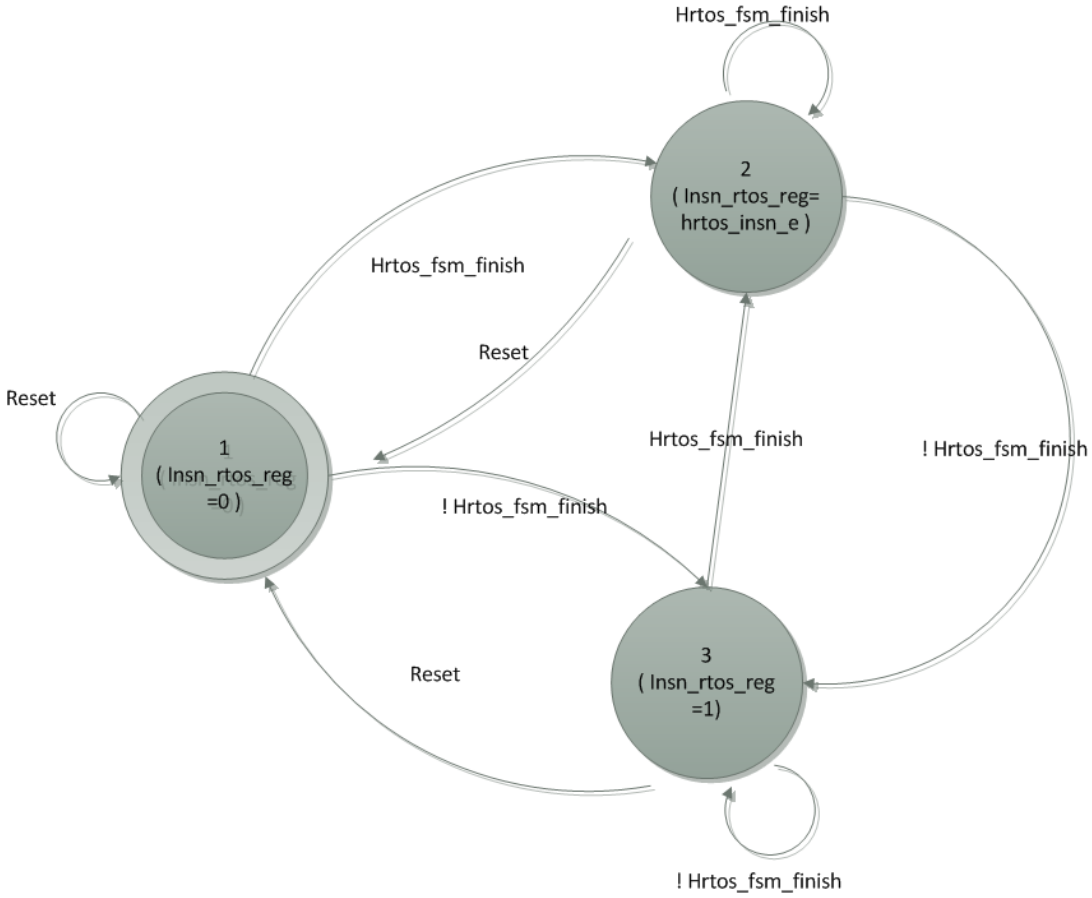


Figura 34: Diagrama de estados do sinal 'insn_rtos_reg'

Quando acontece o 'reset' do sistema 'insn_rtos_reg' passa para o valor '0'. No próximo pulso de relógio como não existe nenhuma FSM ativa, o sinal 'hrtos_fsm_finish' está a '1' e 'insn_rtos_reg' toma o valor de 'hrtos_insn_e'. Caso 'hrtos_insn_e' esteja a '1' (deteção de uma instrução para Hrtos) 'insn_rtos_reg' passa a '1'. Caso 'hrtos_fsm_finish' esteja a '0', 'insn_rtos_reg' é realimentado com o valor '1', senão 'insn_rtos_reg' iniciará outro ciclo.

É de salientar que 'insn_rtos_reg' só passará a '1', um ciclo de relógio depois de 'hrtos_insn_e' ter identificado uma instrução. Este efeito é propositado para deixar o estágio execute do *pipeline* do processador finalizar a sua operação antes que Hrtos comece a operar e retirar o acesso aos seus registos e a memória externa.

4.1.2.3 Ligação do Hrtos aos registos de propósito geral (GPR)

As entradas e as saídas do *register file* terão que estar disponíveis tanto para o Hrtos como para o processador, visto que ambos partilham o mesmo. À entrada do *register file* deve existir *multiplexers* que, consoante a condição de seleção, permitirá ao Hrtos ou ao processador aceder aos seus dados. A Figura 35 representa a ligação entre o Hrtos e o *register file.*

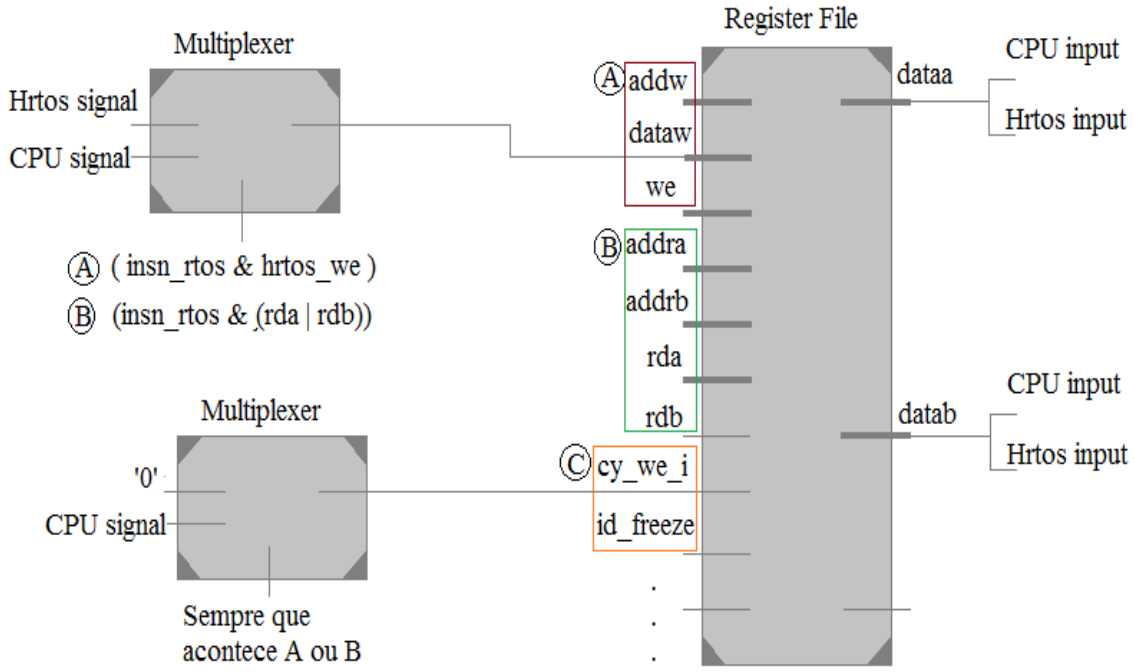


Figura 35: Esquema de ligação entre *Register File* e Hrtos

As entradas do *register file* que Hrtos utilizará para efetuar uma operação de escrita são:

- 'addw' que é o endereço do registo onde será escrita a informação (*wire* de 5bits)
- 'dataw' que é o dado que será escrito no registo escolhido (*wire* de 32 bits)
- 'we' um bit que indicará que se trata de uma operação de escrita

Para a operação de leitura as entradas são:

- 'addra' e 'addrb' que são os endereços do registos a ler, visto que o *register file* é *dual ram* e permite ler dois registo em simultâneo (5bits).
- 'rda' e 'rdb' que são bits de seleção que permitirá ler um registo ('a' ou 'b'), dois ('a' e 'b') ou nenhum (nem 'a' nem 'b').

E as saídas são:

- 'dataa' e 'datab' que são os dados (32bits) devolvido pelos registos selecionado em 'a' e em 'b' respetivamente.

Para cada operação, escrita ou leitura ('A' e 'B' na Figura 35 respetivamente), existe uma condição de seleção diferente. No caso de efetuar uma operação de escrita, o Hrtos terá acesso se 'hrtos_rf_we' e 'insn_rtos' for '1', e no caso de efetuar uma operação de leitura, a condição é se 'insn_rtos' for '1' e tiver um dos sinais 'rda' ou 'rdb' a '1'. Para ambas as operações 'insn_rtos' terá que ser '1', isto porque o Hrtos só poderá aceder ao *register file* caso o CPU esteja em *stall.* A letra 'C', na Figura 35, representa dois sinais que deverão ser postos a zero caso haja uma operação de leitura ou escrita por parte de Hrtos.

É ainda de salientar que os registos de propósito geral, localizados no *register file* (módulo GPRs na Figura 21), apesar de guardarem dados para as funções opera-los, alguns destes registos guardam dados específicos como o *stack pointer,* o *frame pointer* e o *link register* como se pode ver na Tabela 6.

Tabela 6: Registos de Prepósito Geral (GPR) do OR1200

Registo	Dado específico
R0	Sempre com valor 0 (ligado em *hardware*)
R1	SP (*Stack Pointer*) Apontador para o início da pilha da *thread*
R2	FP (*Frame Pointer*) Aponta para uma localização específica na pilha da *thread*
R3	SP, da próxima *thread* a ser executada[3]
R4	SP, da *thread* que está a ser executada[3]
R9	LR (*link register)* guarda o endereço de retorno quando é chamada uma função
R11	RV *(return value)* guarda o valor ou endereço a devolver pela função

[3] No caso de tratar-se da execução da função 'load_context' ou 'context_switch'. Senão são registos normais, usados para guardar dados vindo de operações efetuadas pelo CPU.

4.1.2.4 Ligação e acesso do Hrtos à memória externa de dados

O acesso à memória externa de dados por parte do Hrtos deverá ser efetuado através do barramento Wishbone. Para tal Hrtos deverá possuir um módulo de compatibilidade Wishbone, para o barramento de dados, semelhante ao do processador. Este módulo terá que estar ligado ao *arbiter* do barramento de dados, sendo que deve criar-se novas entradas e saídas para Hrtos no interior do *arbiter Dbus*, ver subcapítulo 3.2.3.

Para ter a certeza que Hrtos tenha prioridade de acesso a memória externa de dados, no interior do *arbiter Dbus* os sinais de Hrtos serão prioritários. Isto acontece tornando Hrtos um *master* prioritário, como se pode ver na Figura 36, Hrtos tornar-se-á o *master* 0 (absoluto) para acessos aos dados.

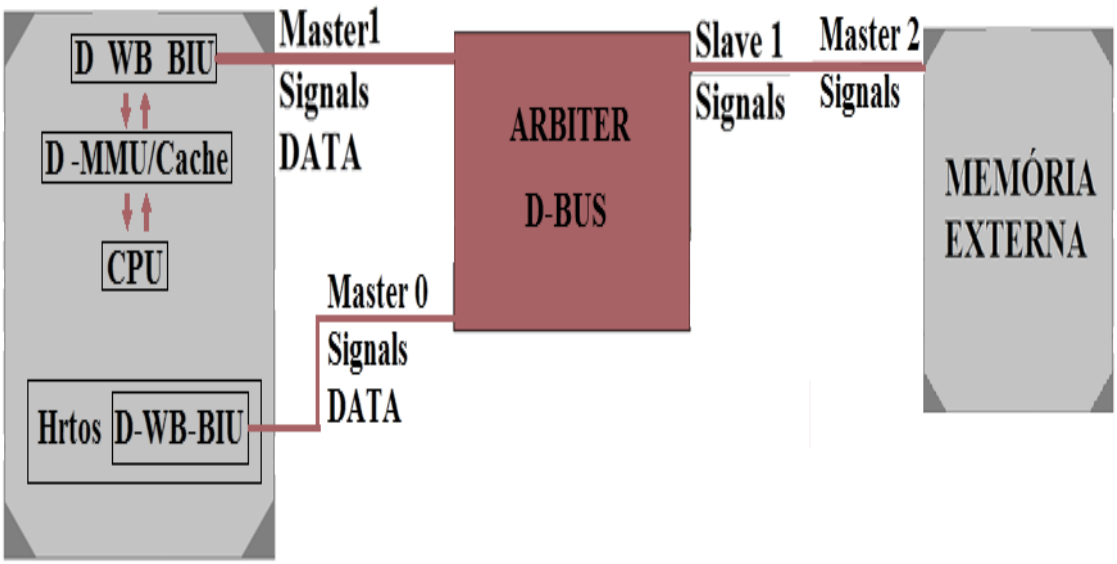


Figura 36: Esquema de ligação entre Hrtos e a memória de dados

4.1.2.5 Ligação e acesso do Hrtos aos registos especiais

O acesso aos registos de propósito especial e ao registo de supervisão será feitos recorrendo à adição de novas entradas e saídas ao módulo 'or1200_sprs'. As novas entradas serão:

- 'hrtos_spr_addr' o endereço do registo a selecionar para ler ou escrever (32bits).
- 'hrtos_spr_read' um bit de sinalização para efetuar uma leitura do registo.
- 'hrtos_spr_write' um bit de sinalização para efetuar escrita no registo.
- 'hrtos_spr_dat_i' o dado inserido por Hrtos no registo selecionado (32bits).

E a única saída será:

- 'hrtos_spr_dat_o' o dado devolvido pelo registo previamente selecionado (32bits).

Para estas entradas e saídas terem efeito é necessário alterar a multiplexagem interna dos sinais deste módulo. Os sinais internos a multiplexar são 'spr_we', 'spr_addr', 'spr_dat_o', dat_i e 'to_wbmux', representados na Figura 37.

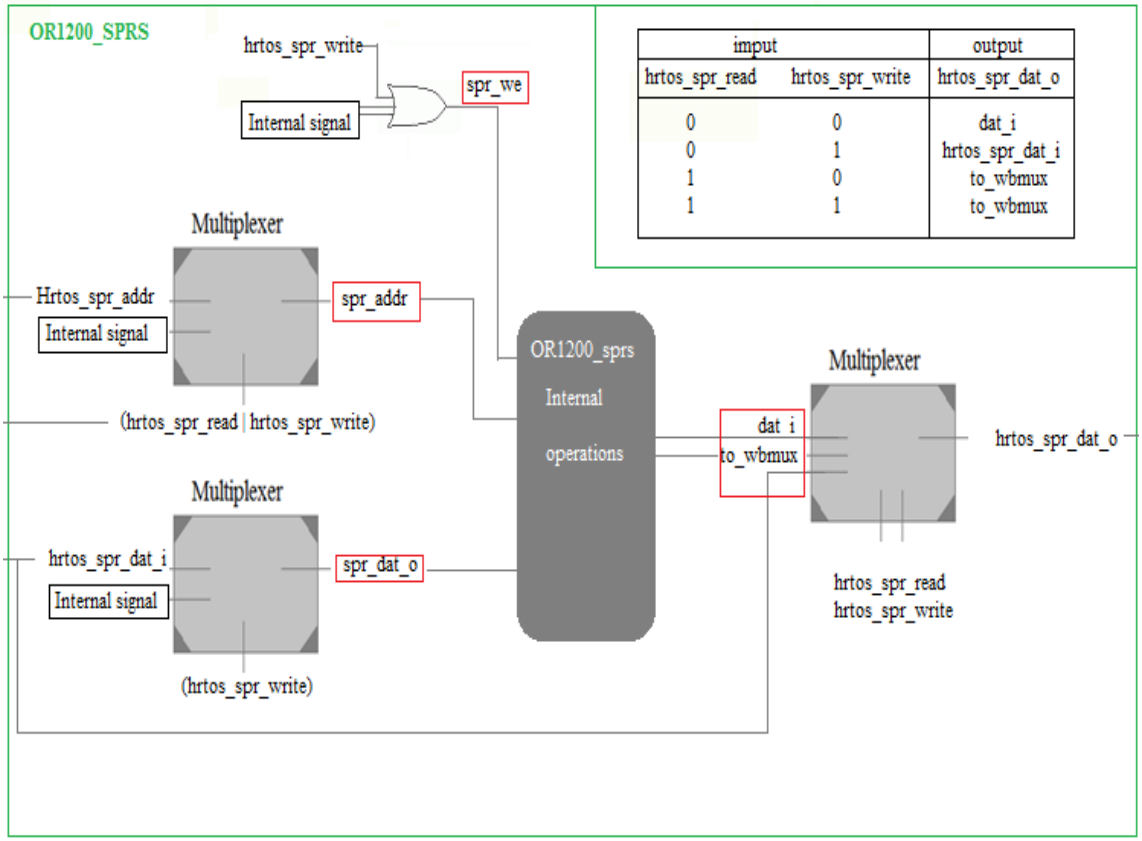


Figura 37: Esquema de ligação entre Hrtos e os registos especiais

4.1.3 Funções a migrar para *hardware*

As funções a migrar para *hardware* serão os métodos das classes 'scheduler', Figura 12, 'thread', 'hardware thread', 'scheduler thread', Figura 13, e 'thread queue', Figura 14. Estas funções serão implementadas recorrendo a máquinas de estados e estarão inseridas no módulo ao qual pertencem, por exemplo um método da classe 'hardware_thread', pertencerá ao módulo 'HTHREAD' no interior de Hrtos. Agora que o encapsulamento com Hrtos está definido, as secções seguintes explicarão como se deverá migrar determinadas funções.

4.1.3.1 Migração da função "Load_Context"

A função 'hal_thread_load_context' também chamada de 'init_context' é um método da classe 'cyg_hardware_thread'. Esta efetua o carregamento dos dados da *thread* a executar, para os registos de propósito geral e para os registos especiais, vindos da memória de dados. Esta função pode ser chamada a executar em dois momentos distintos, possuindo deste modo as duas nomenclaturas que lhe são concedidas:

- 'init_context', quando a função é iniciada após o escalonador ter determinado qual a primeira *thread* a ser executada (início de contexto).
- 'load_context' quando se está num ponto de escalonamento e ocorrer uma comutação de contexto, esta função é chamada por 'context_switch' para efetuar o carregamento dos dados da nova *thread* a executar.

Esta função no eCos™ está implementada em *assembly*. O código desta função ilustrado na Figura 38 e tendo em conta a Tabela 6, verifica-se que esta possui quatro blocos de operações sequenciais distintas. Estas operações são:

1. Receção, através do registo R3, do endereço do *stack pointer* da *thread* e salvaguarda deste no registo R1.
2. Carregamento, usando o valor em R1, dos registos de prepósito geral (R2, R9,R10, R12, R14, R16, R18, R20, R22, R24, R26, R28, R30) da *trhead.*
3. Carregamento, usando o valor em R1, do estado do registo de supervisão da *thread* e junção desse valor com o valor atual do registo.
4. Colocação em R1, do valor do novo *stack pointer* guardado anteriormente pela *thread*.

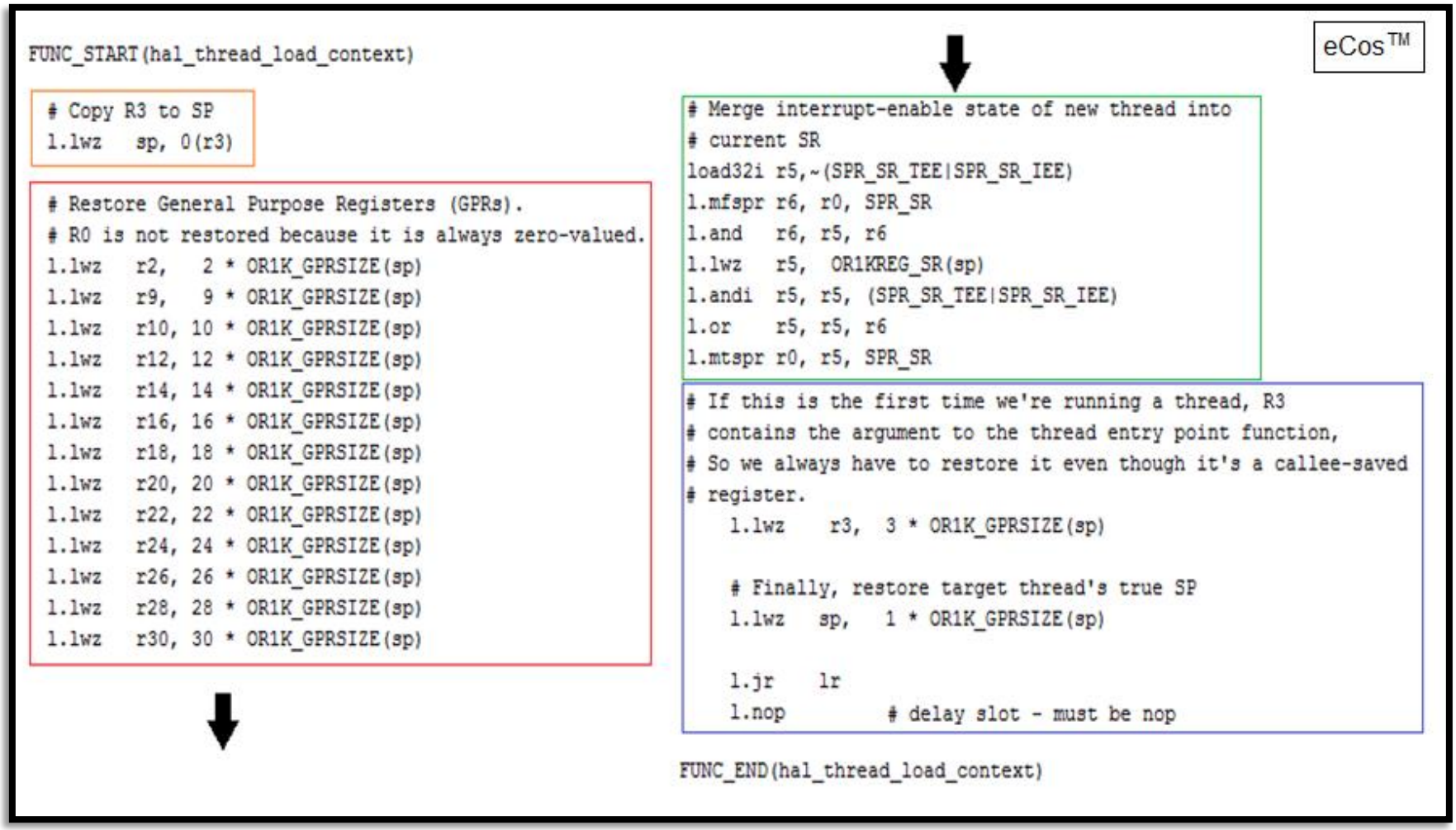

eCos™
FUNC_START(hal_thread_load_context)
# Copy R3 to SP
l.lwz sp, 0(r3)
Restore General Purpose Registers (GPRs).
R0 is not restored because it is always zero-valued.
l.lwz r2, 2 * OR1K_GPRSIZE(sp)
l.lwz r9, 9 * OR1K_GPRSIZE(sp)
l.lwz r10, 10 * OR1K_GPRSIZE(sp)
l.lwz r12, 12 * OR1K_GPRSIZE(sp)
l.lwz r14, 14 * OR1K_GPRSIZE(sp)
l.lwz r16, 16 * OR1K_GPRSIZE(sp)
l.lwz r18, 18 * OR1K_GPRSIZE(sp)
l.lwz r20, 20 * OR1K_GPRSIZE(sp)
l.lwz r22, 22 * OR1K_GPRSIZE(sp)
l.lwz r24, 24 * OR1K_GPRSIZE(sp)
l.lwz r26, 26 * OR1K_GPRSIZE(sp)
l.lwz r28, 28 * OR1K_GPRSIZE(sp)
l.lwz r30, 30 * OR1K_GPRSIZE(sp)
Merge interrupt-enable state of new thread into
current SR
load32i r5,~(SPR_SR_TEE|SPR_SR_IEE)
l.mfspr r6, r0, SPR_SR
l.and r6, r5, r6
l.lwz r5, OR1KREG_SR(sp)
l.andi r5, r5, (SPR_SR_TEE|SPR_SR_IEE)
l.or r5, r5, r6
l.mtspr r0, r5, SPR_SR
If this is the first time we're running a thread, R3
contains the argument to the thread entry point function,
So we always have to restore it even though it's a callee-saved
register.
l.lwz r3, 3 * OR1K_GPRSIZE(sp)
Finally, restore target thread's true SP
l.lwz sp, 1 * OR1K_GPRSIZE(sp)
l.jr lr
l.nop # delay slot - must be nop
FUNC_END(hal_thread_load_context)

Figura 38: Código da função 'hal_thread_load_context' em *software.*

Após a análise das operações que esta função efetua em *software*, contabilizou-se um total de 17 acessos à memória externa para leitura de dados. Este tempo de acesso a memória não poderá ser diminuído, mesmo passando a efetuar a função em *hardware*, isto porque são pedidos individuais e impossíveis de serem efetuados em simultâneo. A Figura 39 representa a nova função que substituirá o código presente na Figura 38, assim como a instrução que iniciará a máquina de estados para o 'load_context'.

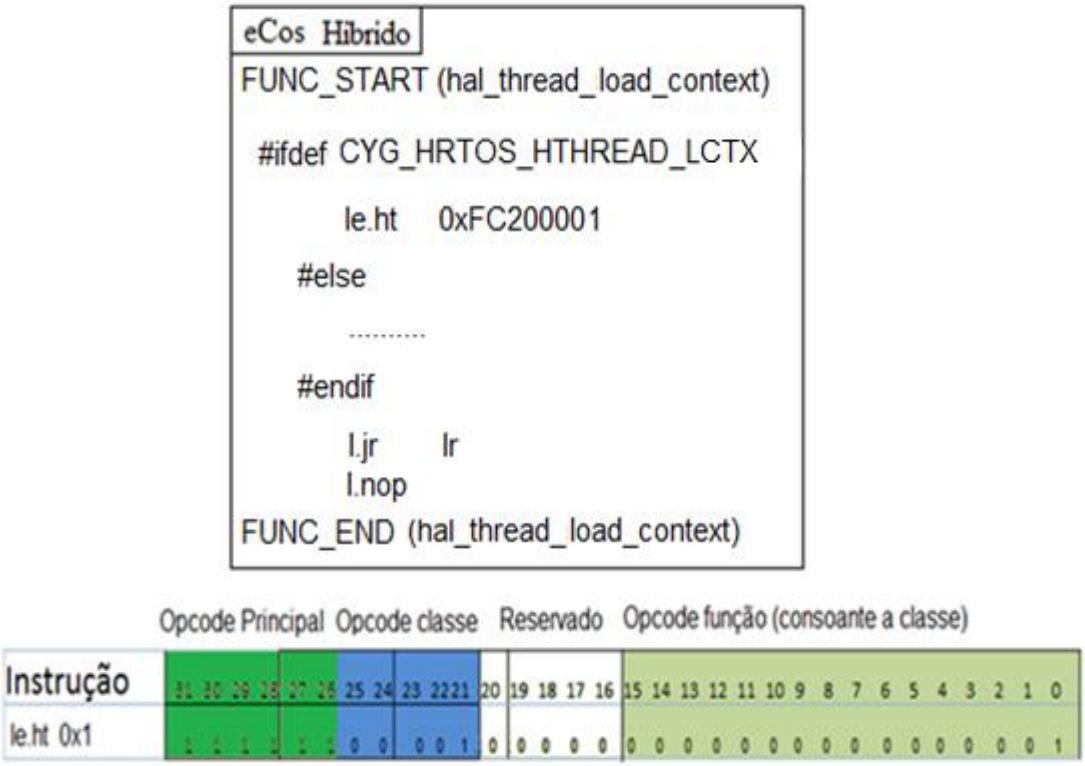

eCos Híbrido
FUNC_START (hal_thread_load_context)
#ifdef CYG_HRTOS_HTHREAD_LCTX
l.ht 0xFC200001
#else
.....
#endif
l.jr lr
l.nop
FUNC_END (hal_thread_load_context)
Opcode Principal
Opcode classe
Reservado
Opcode função (consoante a classe)
Instrução
31 30 29 28 27 26 25 24 23 2221 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
le.ht 0x1
1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Figura 39: Instrução para ativar a função 'load_context' em *hardware*

Para implementar a função 'load_context' usar-se-á uma máquina de estados com 20 estados, sendo um estado parado, um estado inicial para o arranque, um estado de finalização e os restantes 17 são condicionados pelo acesso à memória e receção dos dados. Em simultâneo serão efetuadas a operação de junção dos valores do registo de supervisão e a salvaguarda dos valores recebidos/calculados nos registos respetivos. Para diminuir o *overhead* da leitura de R1, que guarda o *stack pointer,* criar-se-á um registo com esse prepósito. A Figura 40 representa a máquina de estados, que implementará a função 'load_context' em *hardware*.

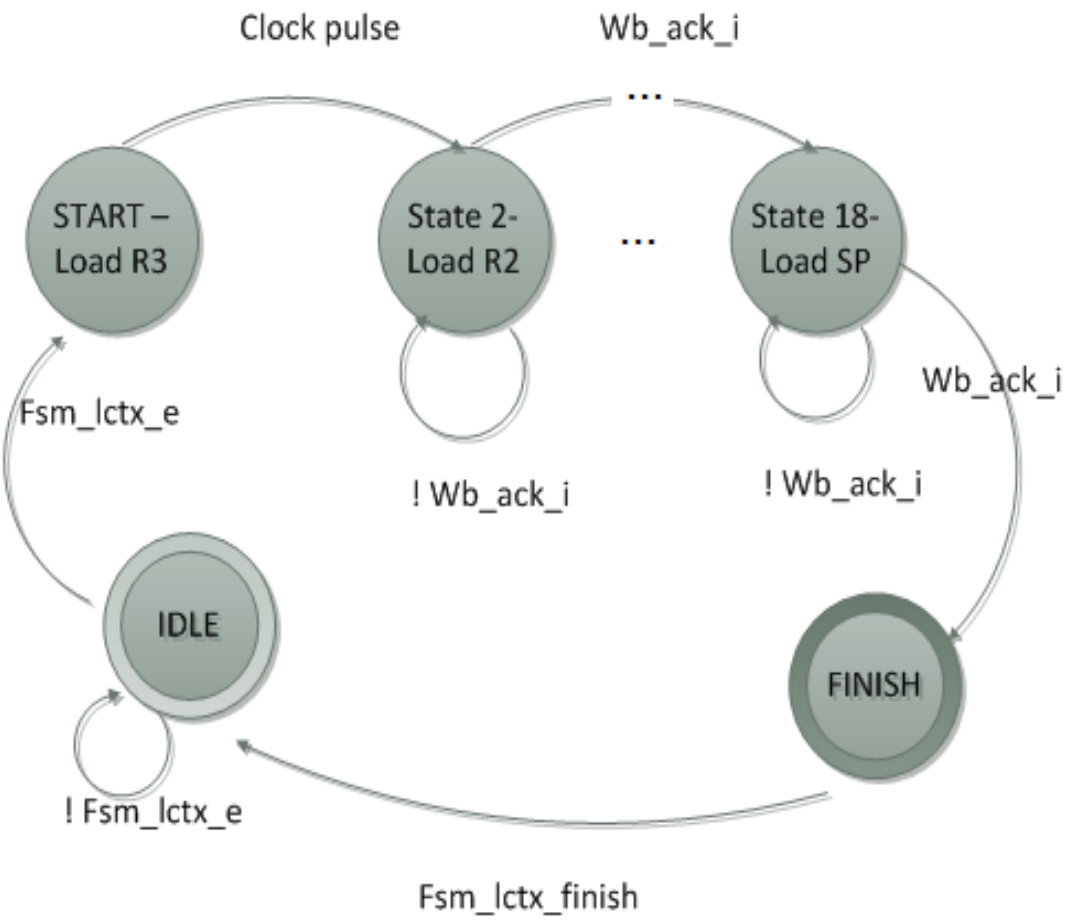


Figura 40: Diagrama de estados da função 'load_context' em *hardware*.

4.1.3.2 Migração da Função "Context_switch"

A função 'hal_thread_load_context' é um método da classe 'cyg_hardware_thread' e está implementada em *assembly*. Esta função opera em duas fases sequenciais:

1. Salvaguarda dos dados da *thread* que estava a executar.
2. Carregamento dos dados da *thread* que irá executar.

A primeira fase, que é o código em *software* representada na Figura 41, realiza a salvaguarda dos dados da *thread*. A segunda fase é efetuada pela chamada da função 'load_context' presente na Figura 38.

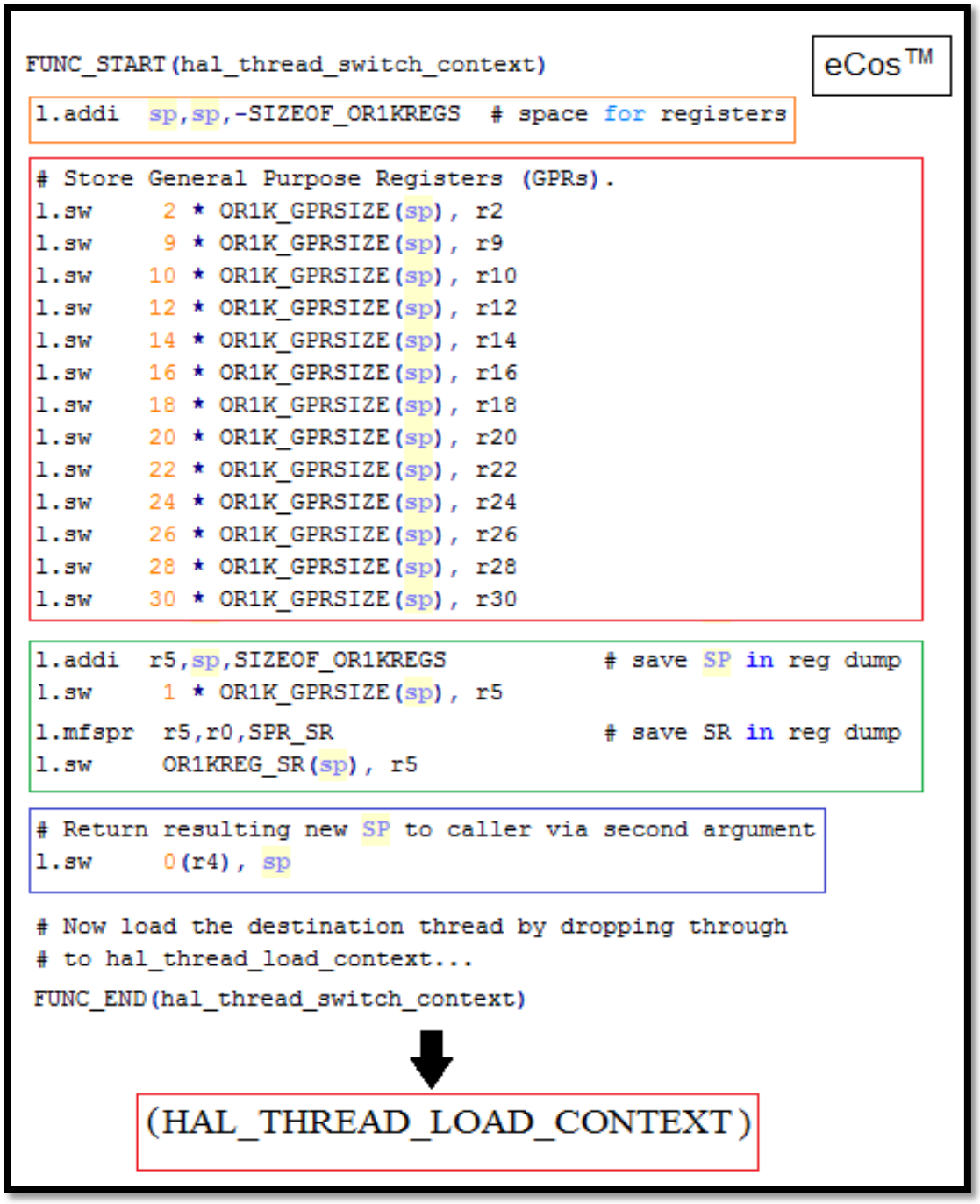

FUNC_START(hal_thread_switch_context)
eCos™
l.addi sp,sp,-SIZEOF_OR1KREGS # space for registers
Store General Purpose Registers (GPRs).
l.sw 2 * OR1K_GPRSIZE(sp), r2
l.sw 9 * OR1K_GPRSIZE(sp), r9
l.sw 10 * OR1K_GPRSIZE(sp), r10
l.sw 12 * OR1K_GPRSIZE(sp), r12
l.sw 14 * OR1K_GPRSIZE(sp), r14
l.sw 16 * OR1K_GPRSIZE(sp), r16
l.sw 18 * OR1K_GPRSIZE(sp), r18
l.sw 20 * OR1K_GPRSIZE(sp), r20
l.sw 22 * OR1K_GPRSIZE(sp), r22
l.sw 24 * OR1K_GPRSIZE(sp), r24
l.sw 26 * OR1K_GPRSIZE(sp), r26
l.sw 28 * OR1K_GPRSIZE(sp), r28
l.sw 30 * OR1K_GPRSIZE(sp), r30
l.addi r5,sp,SIZEOF_OR1KREGS # save SP in reg dump
l.sw 1 * OR1K_GPRSIZE(sp), r5
l.mfspr r5,r0,SPR_SR # save SR in reg dump
l.sw OR1KREG_SR(sp), r5
Return resulting new SP to caller via second argument
l.sw 0(r4), sp
# Now load the destination thread by dropping through
# to hal_thread_load_context...
FUNC_END(hal_thread_switch_context)
(HAL_THREAD_LOAD_CONTEXT)

Figura 41: Código da função 'hal_thread_context_switch' em *software*.

Para migrar esta função para *hardware* operar-se-á em duas fases também, a primeira será composta por uma máquina de estados com pelo menos 19 estados compostos por um estado *idle,* um estado de início e outro de fim e 16 outros obrigatórios, devido aos acessos para escrita na memória de dados. Em simultâneo efetuar-se-á as operações necessárias aos dados para este

serem escritos corretamente. A segunda fase será iniciada com a ativação da FSM 'load_context', que efetuará o carregamento do contexto, como referido em 4.1.3.1.

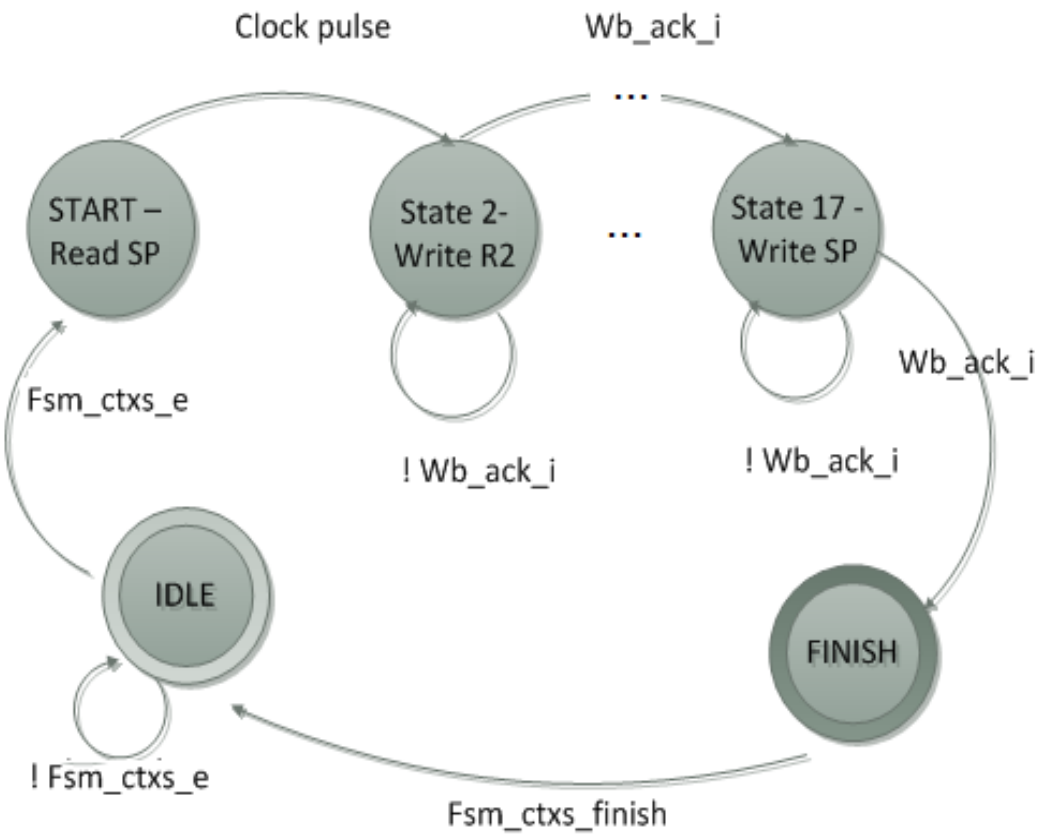


Figura 42: Diagrama de estados da função 'context_switch' em *hardware*.

A instrução que será responsável por desencadear a execução da função 'switch_context' por *hardware* está representada na Figura 43. Esta instrução substituirá o código apresentado na Figura 41. No entanto será necessário acrescentar na biblioteca hibrida duas instruções que serão a instrução de salto com o uso do *link register* e uma instrução 'NOP'. Esta adição será feita porque 'context_switch' em *hardware* chamará 'load_context' em *hardware* sem que o processador tenha conhecimento, logo se esta adição não fosse efetuada, o CPU executaria a próxima instrução, que neste caso seria 'le.ht 0x1'. A nível de execução não haveria diferença, visto que efetuaria o mesmo resultado porque chamaria a função 'load_context', no entanto aumentaria o *overhead* desta operação de dois pulsos de relógio, isto porque 'insn_rtos' viria a zero e voltaria novamente ao valor '1'.

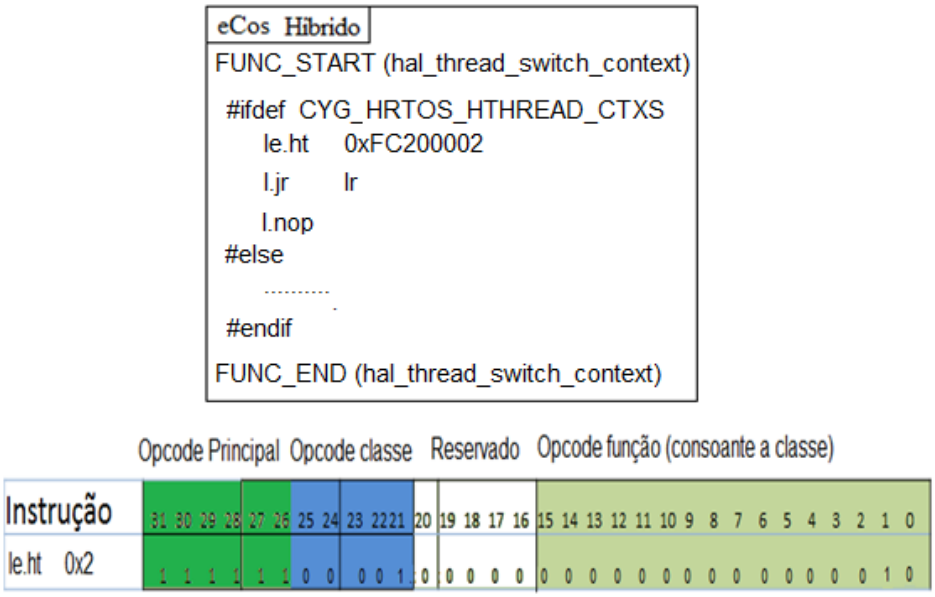


Figura 43: Instrução para ativar 'context_switch' em *hardware*

Como a FSM 'context_switch' terá que ativar a FSM 'load_context' será necessário criar uma condição que permite ativar o *enable* da função 'load_context', após 'context_switch' terminar. Esta passará pela utilização de uma máquina de estados representada na Figura 44.

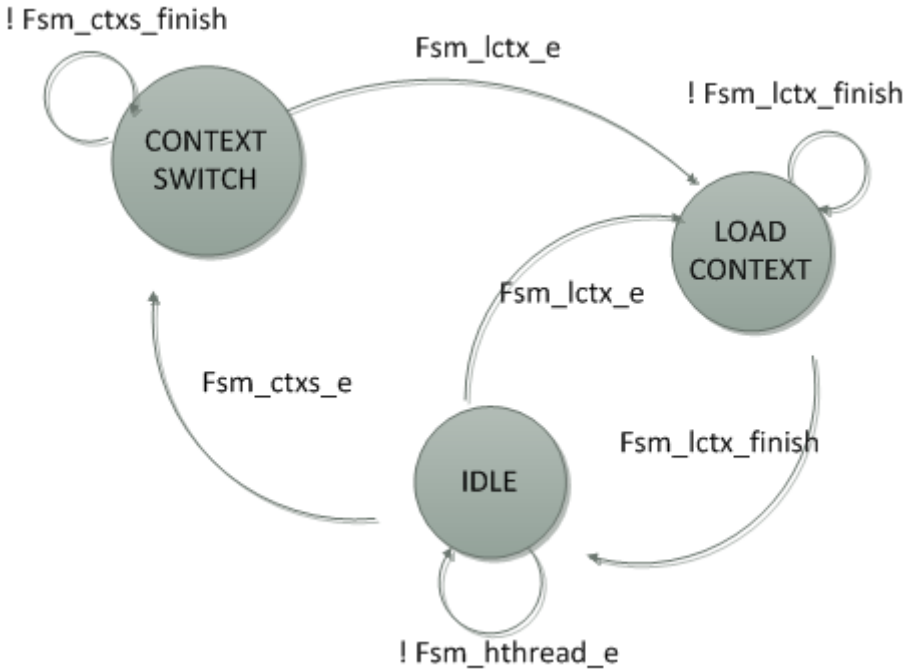


Figura 44: Diagrama de estado para troca entre funções

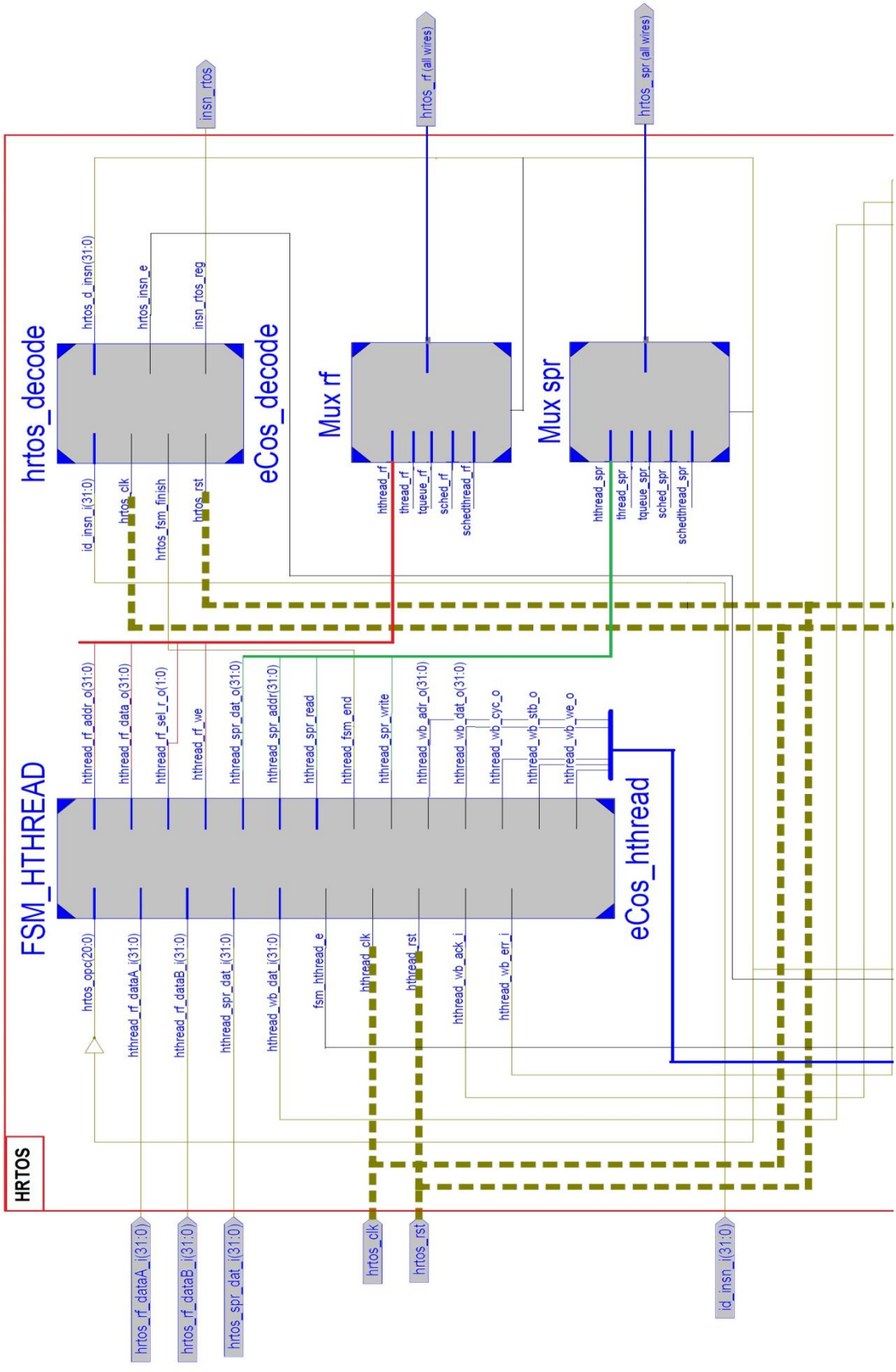
4.2 Implementação

Neste subcapítulo estão apresentados alguns esquemáticos da implementação final resultante do *gateware* efetuado. Estes esquemáticos foram obtidos através da ferramenta RTL-*viewer* presente no IDE da Xilinx, após ter sintetizado o código de descrição de *hardware* desenvolvido. Estes esquemáticos representam em *gate-level* os seguintes módulos:

- Hrtos, Figura 45, ilustra a composição de Hrtos e a ligações entre os seus módulos. Estes módulos internos são a unidade de 'decode', a unidade 'hthread' e a interface com o barramento Wishbone.

- Hthread, Figura 46, ilustra a composição deste módulo que agrupa os métodos presentes na classe 'hardware_thread', neste estão contidas as FSM das funções 'load_context' e 'context_switch'.

- CPU com Hrtos, Figura 47, ilustra as ligações entre o CPU e o *hardware* RTOS desenvolvido no interior do processador. Estas ligações entre estes dois módulos passam pela ligação ao *register File, special purpose register* e ao módulo de *freeze* com o *wire* 'insn_rtos'.


HRTOS
FSM_HTHREAD
eCos_hthread
hrtos_decode
eCos_decode
Mux rf
Mux spr
insn_rtos
hrtos_rf (all wires)
hrtos_ spr (all wires)
hrtos_d_insn(31:0)
hrtos_insn_e
insn_rtos_reg
id_insn_i(31:0)
hrtos_clk
hrtos_fsm_finish
hrtos_rst
hthread_rf
thread_rf
tqueue_rf
sched_rf
schedthread_rf
hthread_spr
thread_spr
tqueue_spr
sched_spr
schedthread_spr
hthread_rf_addr_o(31:0)
hthread_rf_data_o(31:0)
hthread_rf_sel_r_o(1:0)
hthread_rf_we
hthread_spr_dat_o(31:0)
hthread_spr_addr(31:0)
hthread_spr_read
hthread_fsm_end
hthread_spr_write
hthread_wb_adr_o(31:0)
hthread_wb_dat_o(31:0)
hthread_wb_cyc_o
hthread_wb_stb_o
hthread_wb_we_o
hrtos_opc(20:0)
hthread_rf_dataA_i(31:0)
hthread_rf_dataB_i(31:0)
hthread_spr_dat_i(31:0)
hthread_wb_dat_i(31:0)
fsm_hthread_e
hthread_clk
hthread_rst
hthread_wb_ack_i
hthread_wb_err_i
hrtos_rf_dataA_i(31:0)
hrtos_rf_dataB_i(31:0)
hrtos_spr_dat_i(31:0)
hrtos_clk
hrtos_rst
id_insn_i(31:0)

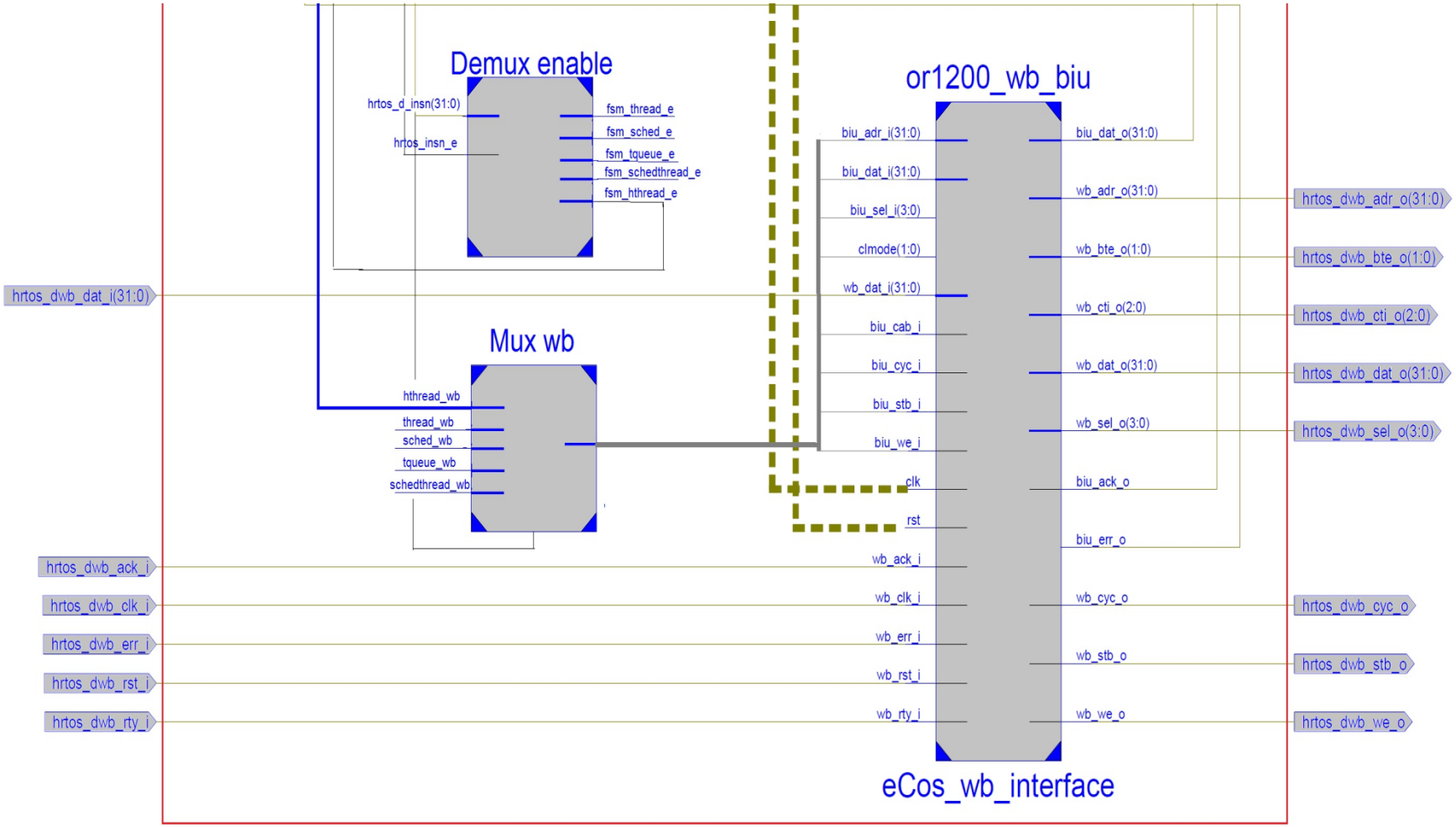


Figura 45: RTL – Hrtos

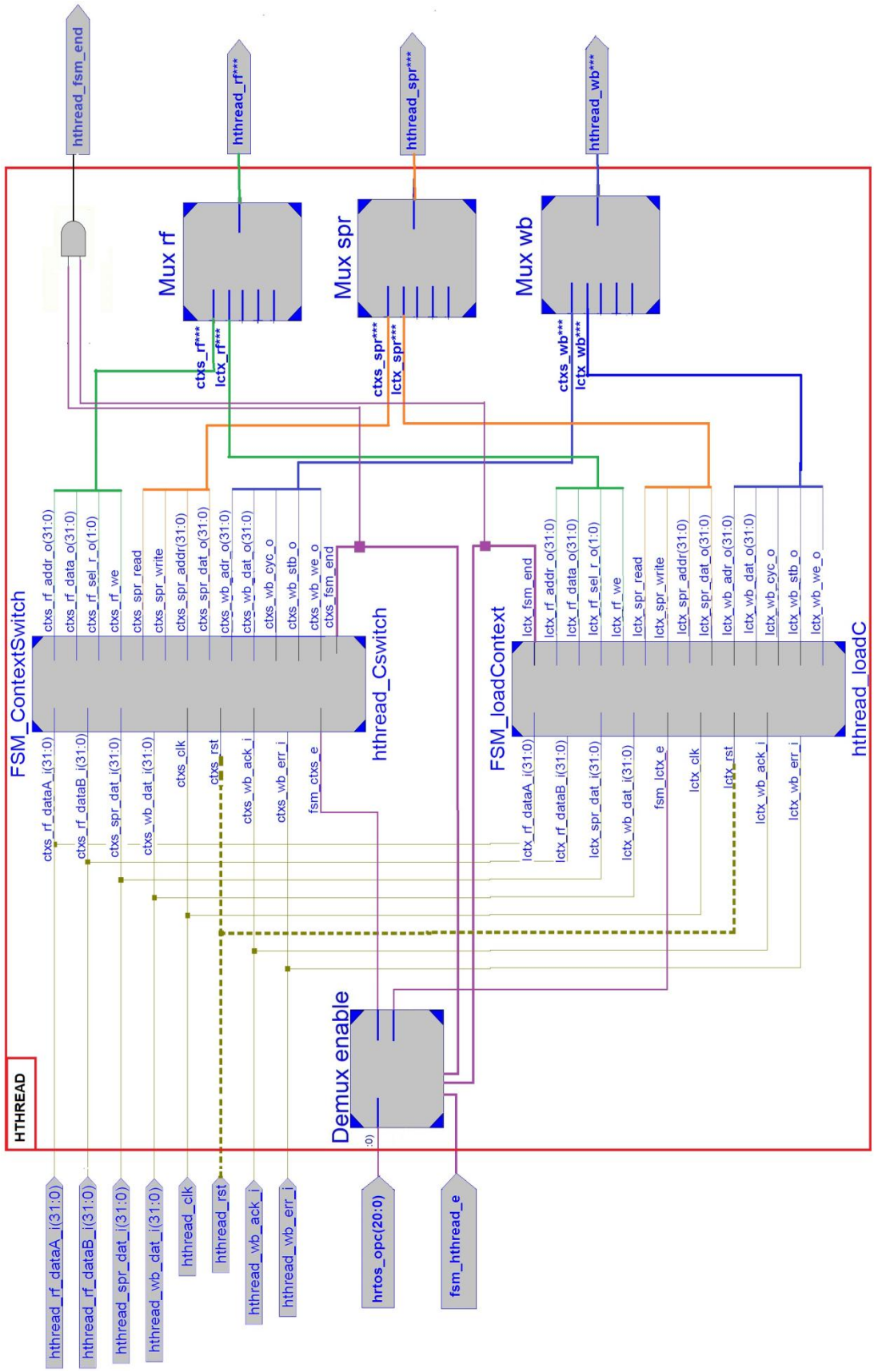


Figura 46: RTL –HTHREAD

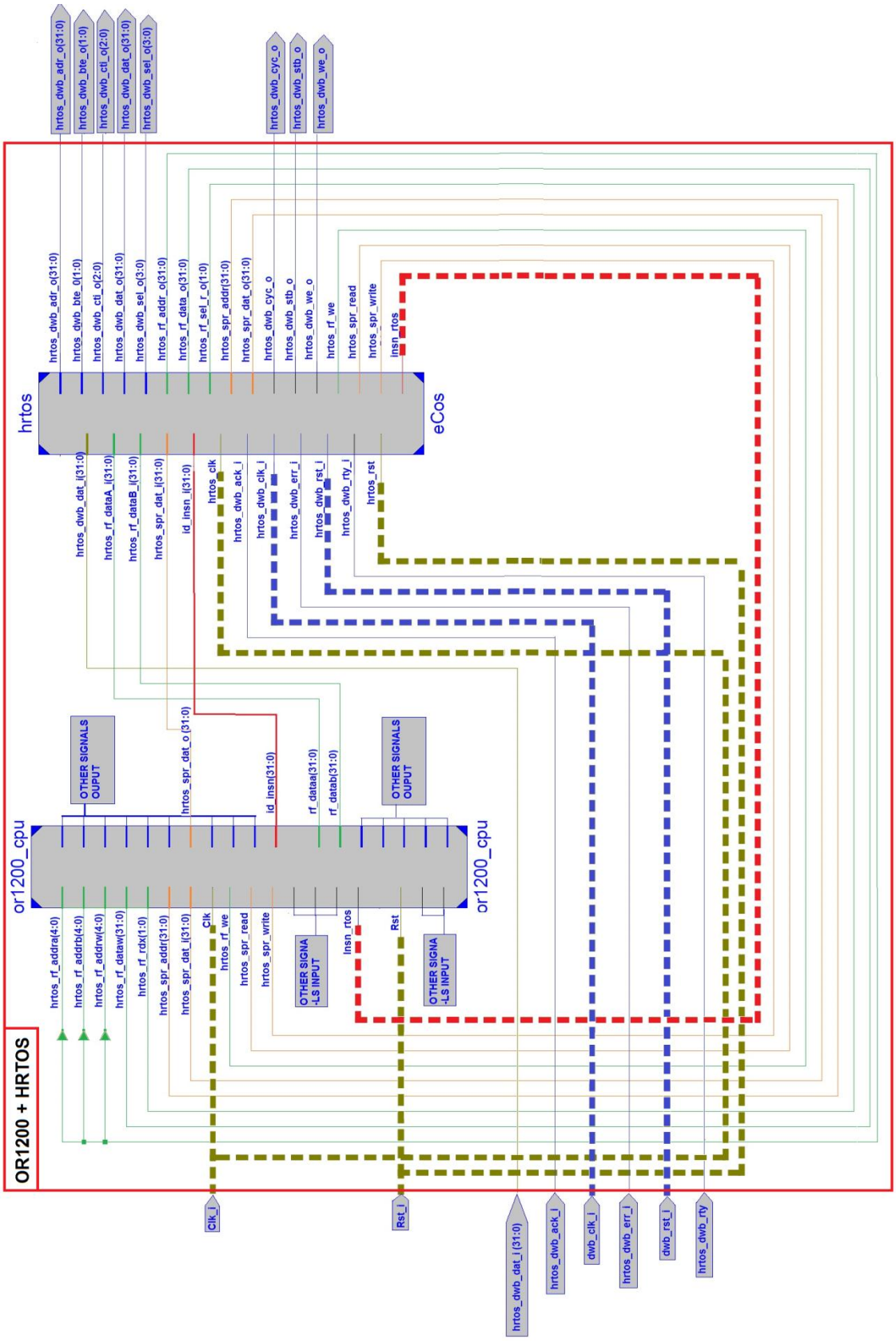


Figura 47: RTL - CPU e Hrtos

Capítulo 5

Resultados Experimentais

Este capítulo demonstra quais foram as metodologias de teste aplicadas e as ferramentas usadas para a realização dos testes. Serão apresentados os resultados obtidos e para finalizar será efetuada uma análise a esses resultados.

5.1 Metodologias de Teste

Os testes para a verificação do funcionamento de Hrtos, foram realizados usando os *benchmarks* do 'MIBench', esta *suite* de *benchmarks* é *open-source* e serve para medir e comparar a performance entre vários sistemas. Como dispositivo usou-se a placa XC5VLX110T-FF1136 da família Virtex5, em que o processador Or1200 estava sintetizado para funcionar com um clock de 66Mhz.

Foram usados dois *benchmarks* do 'Mibench', o 'bitcnt' e o 'stringsearch', cada um realizado numa *thread.* Para ambos os *benchmarks* correrem no mesmo executável foi necessário criar um ficheiro '.c' com a função 'cyg_user_start' que cria as *threads* com as funções 'bitcnt' e 'stringsearch' respetivamente. Este ficheiro está representado na Figura 48.

```c
#include <cyg/kernel/kapi.h>

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* now declare (and allocate space for) some kernel objects,
   like the two threads we will use */
cyg_thread thread_s[2];         /* space for two thread objects */

char stack[2][4096];            /* space for two 4K stacks */

/* now the handles for the threads */
cyg_handle_t threadA, threadB;

/* and now variables for the procedure which is the thread */
cyg_thread_entry_t StrSearch, Bitcnt;

/* and now a mutex to protect calls to the C library */
cyg_mutex_t cliblock;

/* we install our own startup routine which sets up threads */
void cyg_user_start(void)
{
  printf("Entering TEST Multi_Threads' <OR1200 + eCos>\n");

  cyg_mutex_init(&cliblock);

  cyg_thread_create(4, StrSearch, (cyg_addrword_t) 0,
                    "Thread A", (void *) stack[0], 4096,
                    &threadA, &thread_s[0]);

  cyg_thread_create(5, Bitcnt, (cyg_addrword_t) 1,
                    "Thread B", (void *) stack[1], 4096,
                    &threadB, &thread_s[1]);

  cyg_thread_resume(threadA);
  cyg_thread_resume(threadB);

}
```

Figura 48: Código principal usado para efetuar as simulações

Após compilar o código acima com os ficheiros dos *benchmark* do 'Mibench', foi necessário converter a aplicação para um ficheiro binário de modo a carregá-lo para a memória do processador. Este passo está descrito no Anexo IV.

Tendo o código máquina sido carregado para a memória do processador foram realizados os testes de aprovação e medição das métricas recorrendo às seguintes ferramentas:

- 'ISim' para medir a *performance* temporal das novas funções e verificar a sua correta execução, em comparação com as funções originais implementadas em *software*.
- 'XPower Analyser', para medição da potência consumida que juntamente com o tempo de execução resultaram no cálculo da energia dissipada, e o número de unidades lógicas usadas (LUTs, Flip-Flop e *Block ram*).
- Sintetização na placa XC5VLX110T, usando o 'iMPACT' para efetuar um teste em tempo real do correto funcionamento das funções migradas, ver Anexo V.

Estas três ferramentas, 'iMPACT', 'ISim' e 'XPower Analyser', fazem parte do conjunto do ISE da Xilinx.

5.2 Testes e Resultados

Os testes realizados foram efetuados com a comparação entre processador com funções do eCos™ *standard* e com funções realizadas em *hardware* por Hrtos. Estes testes e resultados estão divididos em teste de funcionamento, teste de *performance* temporal, resultados das unidades lógicas consumidas e energia dissipada.

5.2.1 Teste de funcionamento

Os testes de funcionamento serviram para verificar se as funcionalidades implementadas em *hardware* funcionam e se o output originado era semelhante ao output das funções originais. A Figura 49 mostra o funcionamento de Hrtos no momento de efetuar um 'context_switch',

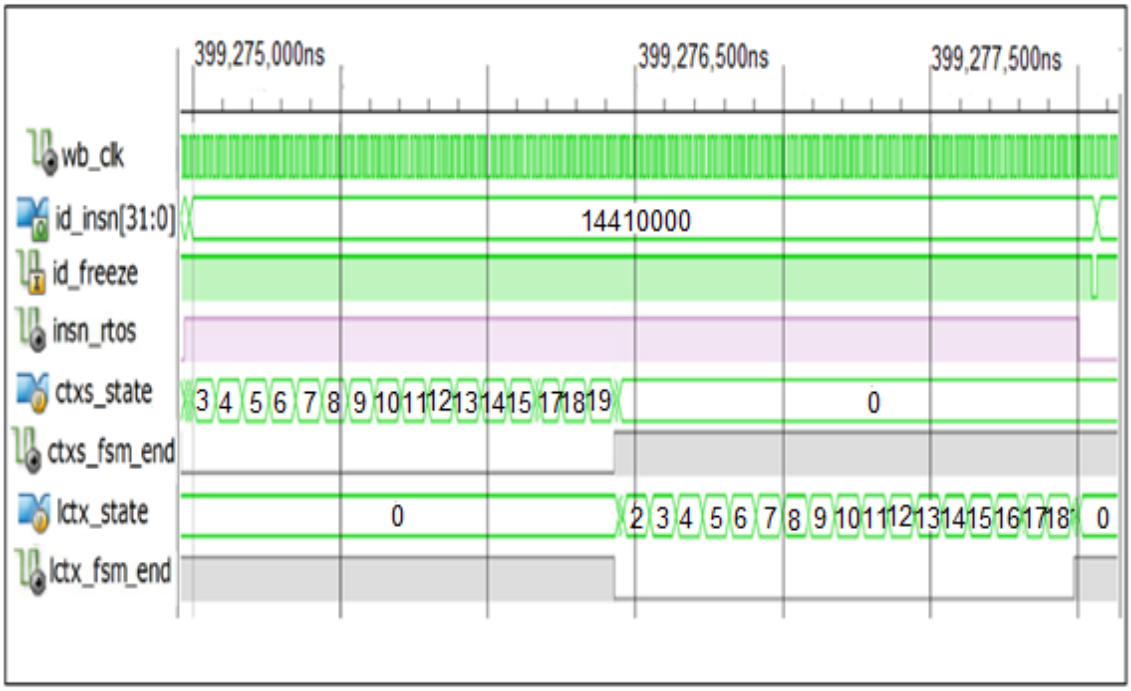


Figura 49: Execução da função 'context_switch' pelo Hrtos

O output originado pela função 'Load_context' e pela função 'context_switch' com e sem Hrtos, num determinado momento em que finalizou a sua execução está representado na Tabela 7.

Tabela 7: Output da função 'Load_context' e 'switch_context'

	Função 'load_context'			Função 'switch_context'		
Registos	OR1200(eCos™ *standard*)	OR1200+Hrtos (eCos hibrido)	Verificação	OR1200(eCos™ *standard*)	OR1200+Hrtos (eCos hibrido)	Verificação
SR	0x8207	0x8207	✔	0x8207	0x8207	✔
R1	0x3a924	0x3a934	✘	0x3b974	0x3b984	✘
R2	0x3a924	0x3a934	✘	0x3b974	0x3b984	✘
R3	0x3b944	0x3b944	✔	0x3ba20	0x3ba20	✔
R5	0x8207	0x8207	✔	0x8207	0x8207	✔
R6	0x8201	0x8201	✔	0x8201	0x8201	✔
R9	0x11C10	0x1194C	✘	0x11c5c	0x11998	✘
R10	0x404000a	0x404000a	✔	0x303000a	0x303000a	✔
R12	0x404000c	0x404000c	✔	0x303000c	0x303000c	✔
R14	0x404000e	0x404000e	✔	0x303000e	0x303000e	✔
R16	0x4040010	0x4040010	✔	0x3030010	0x3030010	✔
R18	0x4040012	0x4040012	✔	0x3030012	0x3030012	✔
R20	0x4040014	0x4040014	✔	0x3030014	0x3030014	✔
R22	0x4040016	0x4040016	✔	0x3030016	0x3030016	✔
R24	0x4040018	0x4040018	✔	0x3030018	0x3030018	✔
R26	0x404001a	0x404001a	✔	0x303001a	0x303001a	✔
R28	0x404001c	0x404001c	✔	0x303000c	0x303000c	✔
R30	0x404001e	0x404001e	✔	0x303000a	0x303000a	✔

Por fim a Figura 50 representa o que acontece quando uma interrupção externa ocorre, enquanto Hrtos estiver a operar.

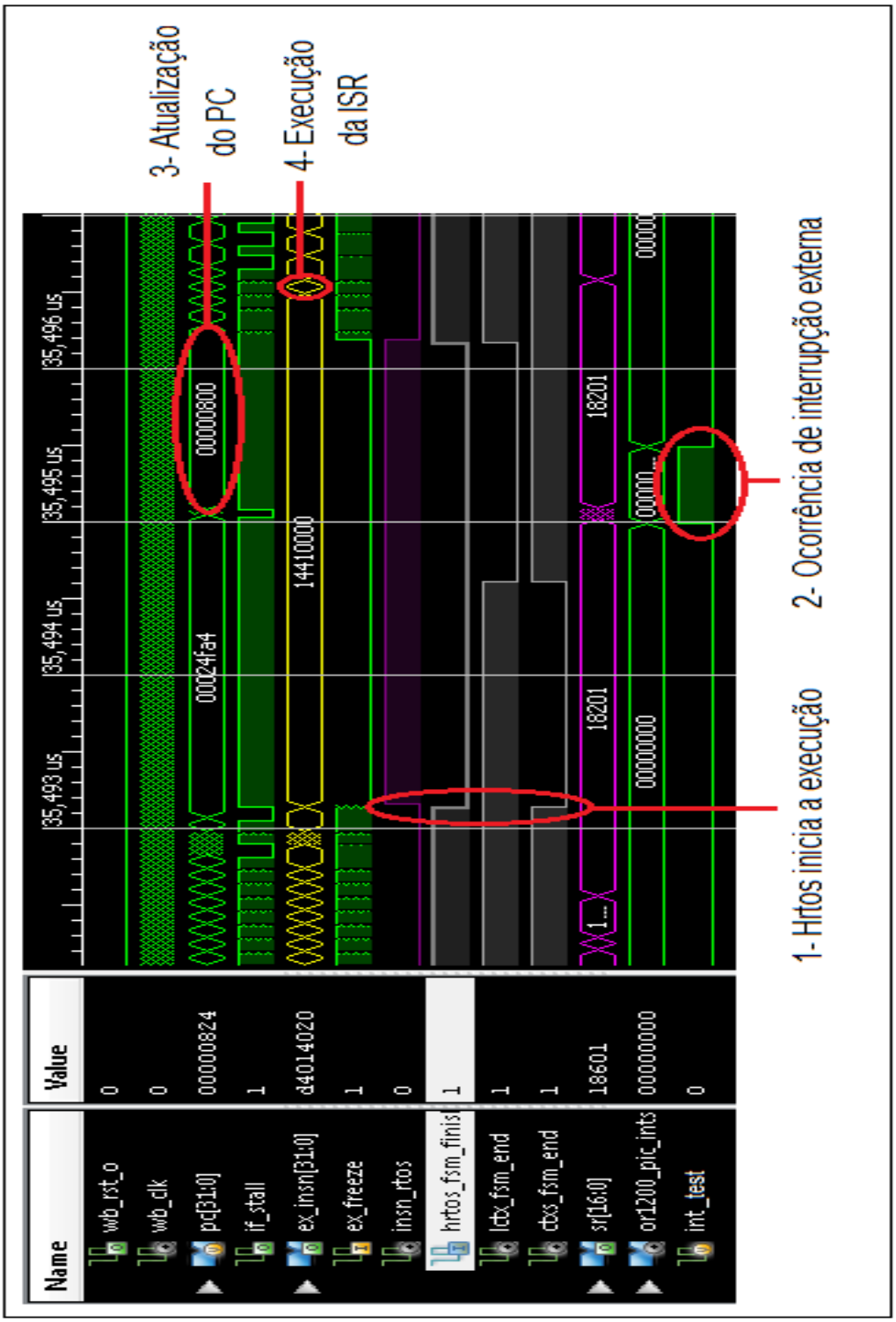


Figura 50: Ocorrência de interrupção quando Hrtos está em execução

5.2.2 Testes de desempenho temporal

Os testes de desempenho temporal foram realizados para obter valores de comparação entre o OR1200 *standard* e OR1200 com Hrtos, quanto ao:

- Tempo de execução da função 'Load_Context' e da função 'Switch_Context', Figura 51;
- Tempo de finalização do programa, Figura 52, usando o escalonador,
  - A. Bitmap: com as duas *threads* com prioridades diferentes;
  - B. Mlqueue: com as duas *threads* com prioridades iguais, onde existe *timeslice* de 30ms.

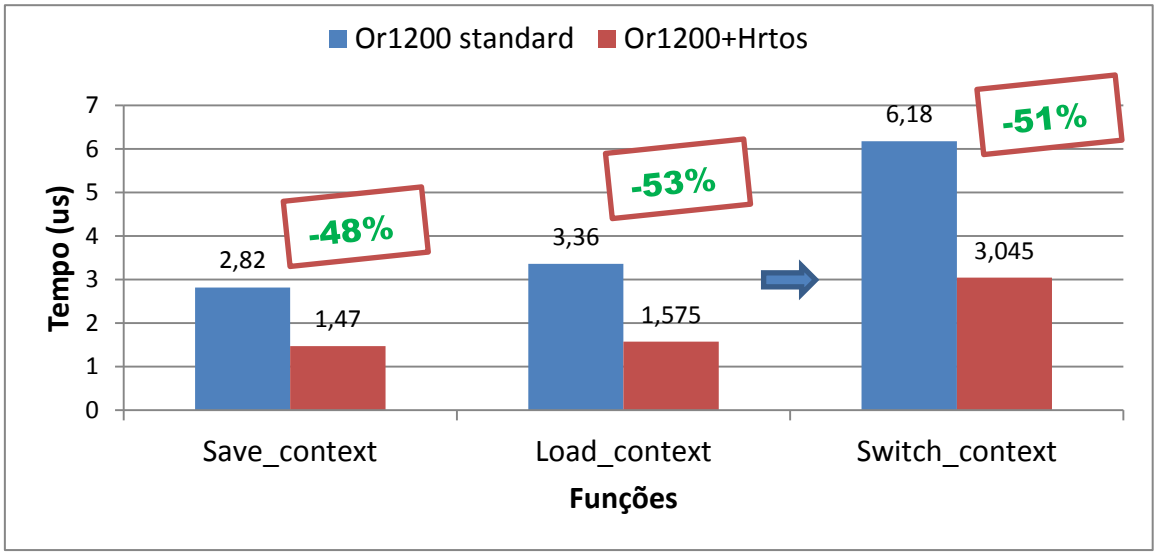


Figura 51: Tempo de execução das funções 'load_context' e 'context_switch' com e sem Hrtos

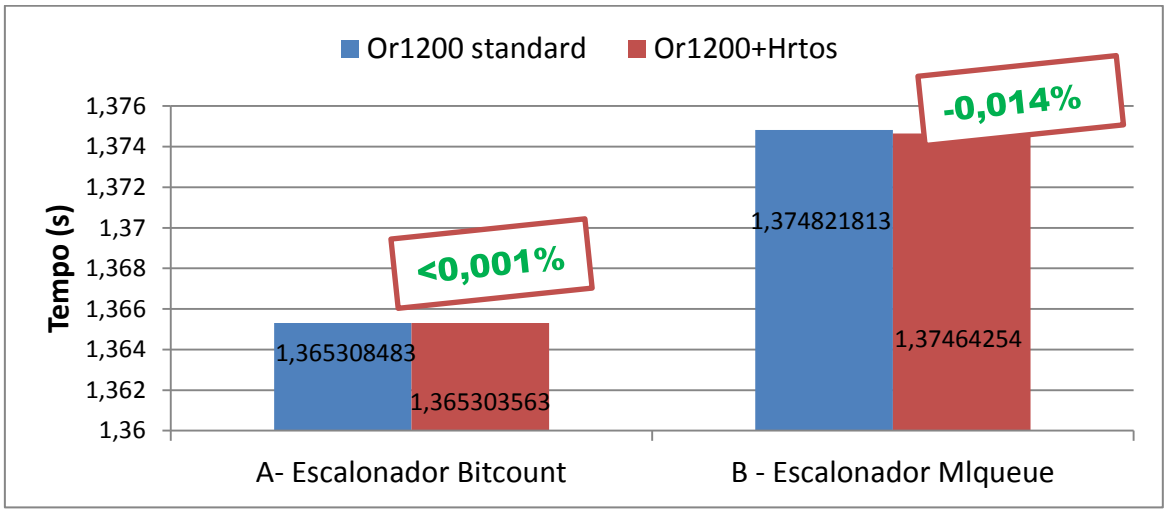


Figura 52: Tempo de finalização do programa usando o escalonador Bitmap e Mlqueue.

5.2.3 Resultados das unidades lógicas usadas

A placa usada possui um total de 69120 *slice register* e *slice LUTs*. Para efeito de medição efetuou-se a contagem da totalidade de *lookup tables*, *Flip-flop* e *block rams* utilizadas por OR1200 com e sem o Hrtos, Figura 53, e a percentagem de ocupação destas duas configurações na XC5VLX110T, Figura 54.

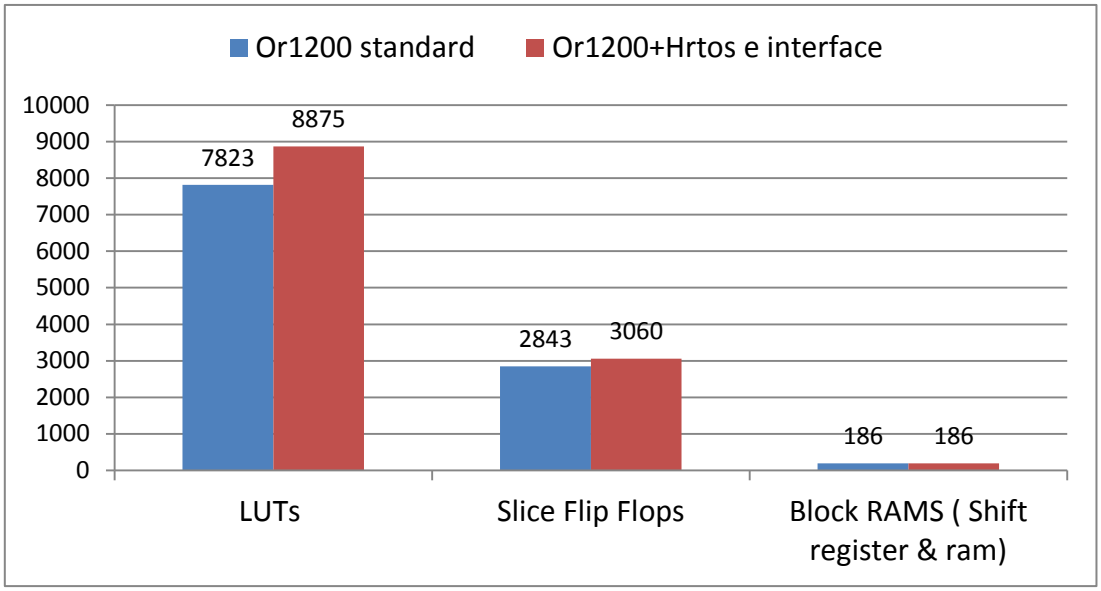


Figura 53: Unidades lógicas totais utilizada por OR1200 com e sem Hrtos

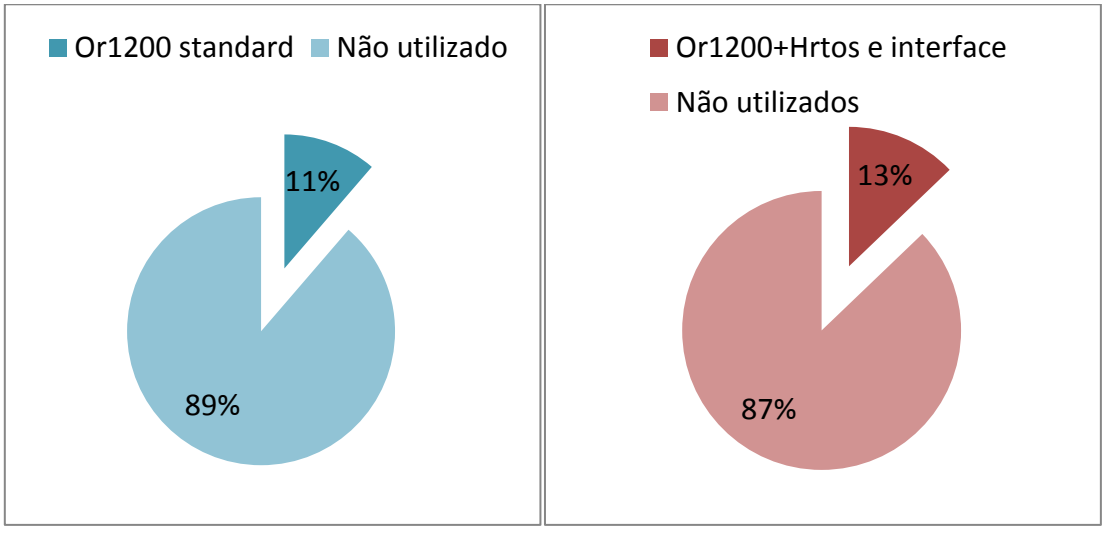


Figura 54: Ocupação percentual de OR1200 com e sem Hrtos na placa

Determinou-se também as unidades lógicas utilizadas somente pelo módulo Hrtos, Figura 55, descriminando os *wires* de interface com os módulos externos e a ocupação percentual das unidades totais de Hrtos pelos seus diversos módulos, Figura 56.

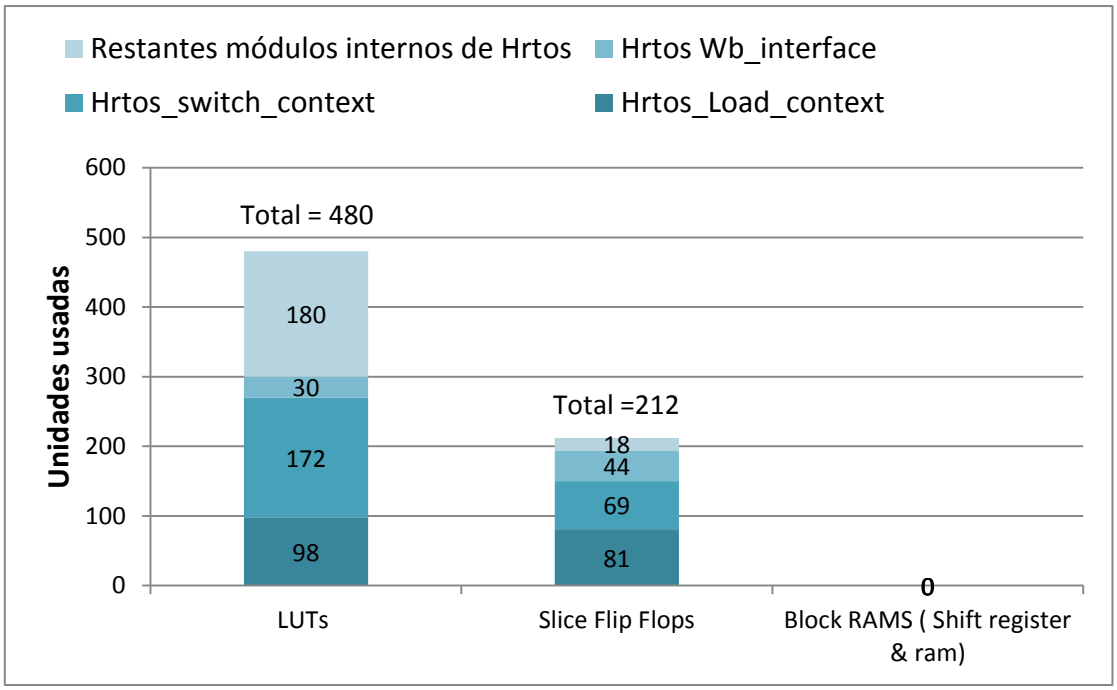


Figura 55: Distribuição por módulos das unidades lógicas utilizadas por Hrtos

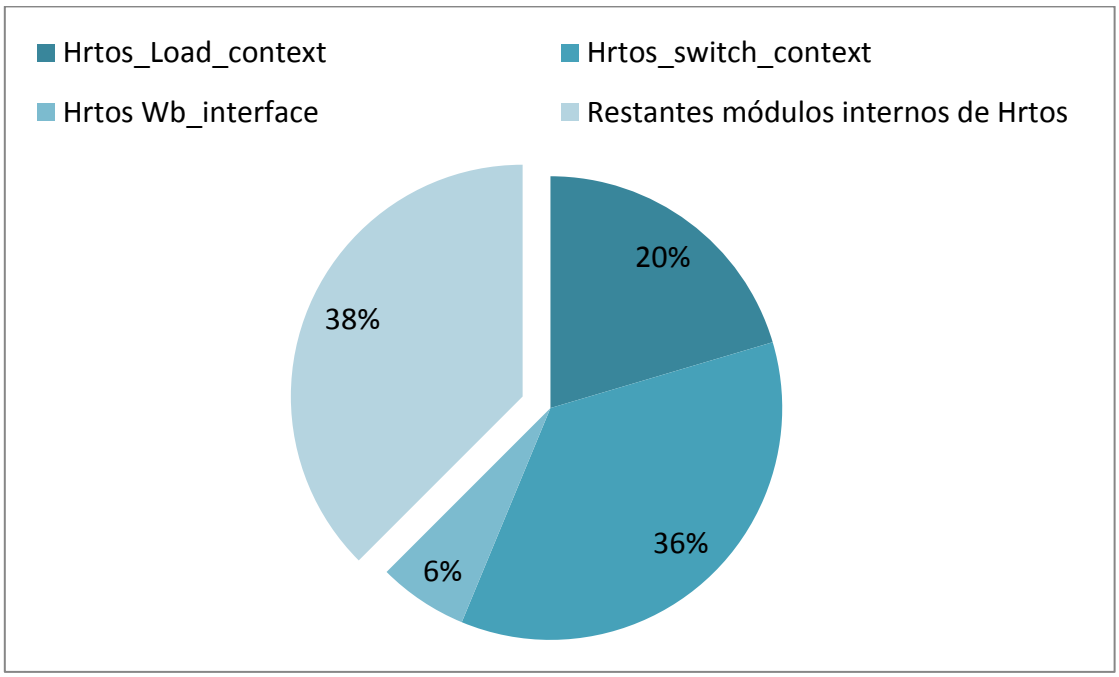


Figura 56: Distribuição percentual das unidades lógicas utilizadas pelos módulos de Hrtos

5.2.4 Resultados da energia dissipada

Para calcular a energia dissipada foi necessário determinar a potência consumida por parte do OR1200 com e sem Hrtos, Figura 57.

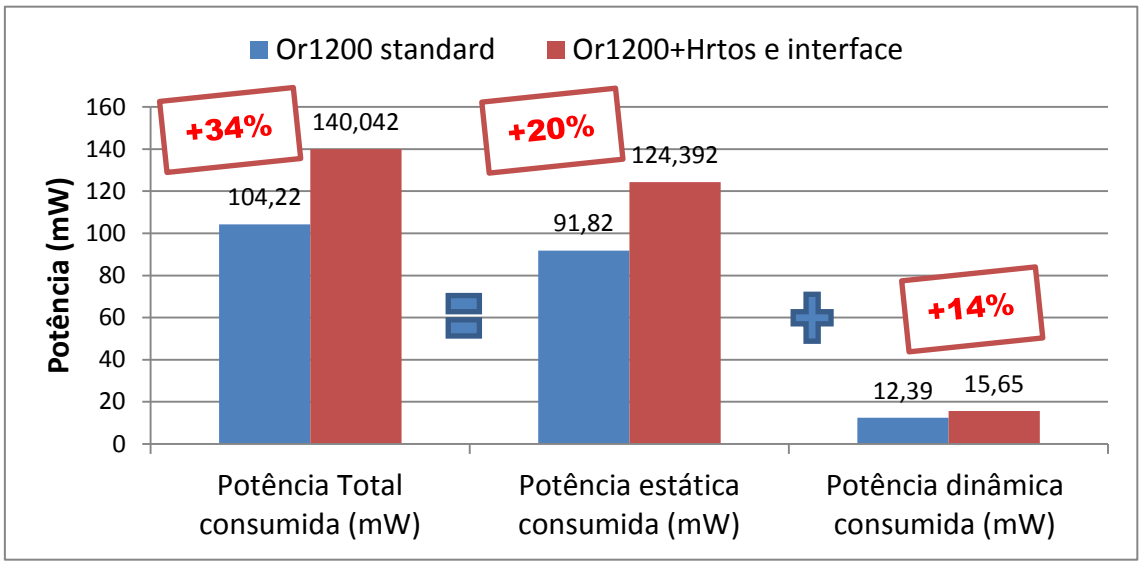


Figura 57: Potência consumida por OR1200 com e sem Hrtos

Com estes dados e os dados obtidos em 5.2.2 efetuou-se o calculo da energia dissipada para:

A. Simulação com o escalonador 'Bitmap' B. Simulação com o escalonador 'Mlqueue'

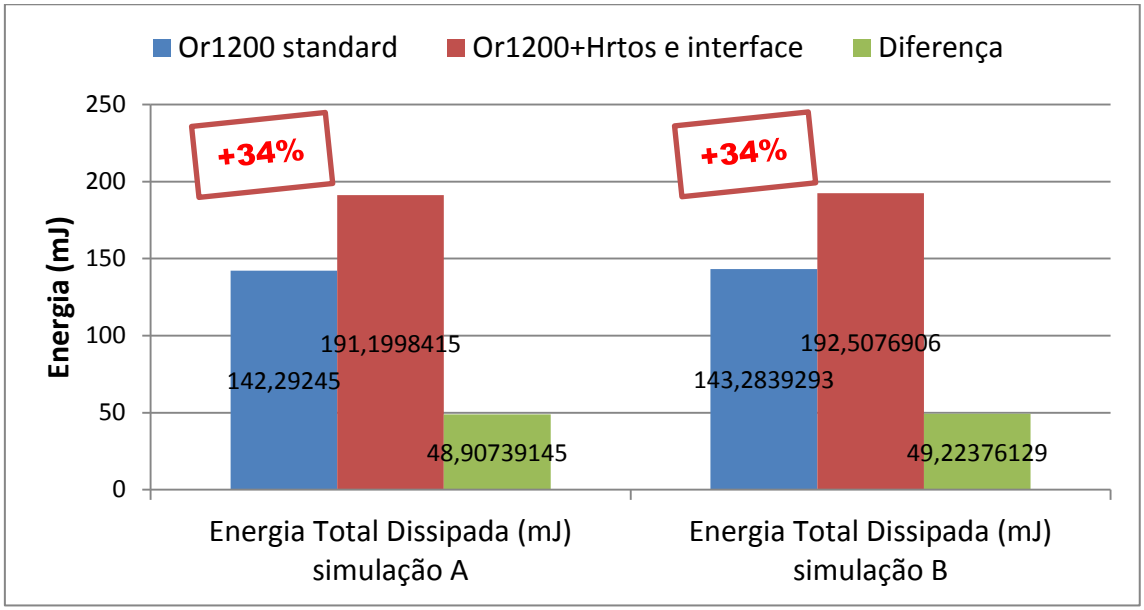


Figura 58: Energia dissipada total por OR1200 com e sem Hrtos,

5.3 Análise dos resultados

Após a realização dos testes, efetuou-se a seguinte análise e discussão dos resultados obtidos:

A. Análise do Funcionamento

Com base na Figura 49, verifica-se que a função em *hardware* 'context_switch' inicia pela máquina de estados para salvaguarda de contexto, 'ctxs_state', sendo que 'hrtos' coloca o sinal de paragem do CPU a '1'. Como foi abordado no subcapítulo 4.1.3.2, a função 'context_switch' ativa 'load_context' após finalizar a salvaguarda de contexto, 'lctxs_state'. Assim que 'load_context' finaliza, Hrtos devolve a execução o CPU, colocando 'insn_rtos' a zero. Esta sequência de operações demonstram que a nível de execução 'context_switch' e 'load_context' operam como era esperado.

A nível de output originado, Tabela 7, verifica-se que tanto para as funções realizadas pelo CPU (função standard do eCos™) como pelas funções realizadas por Hrtos os resultados são iguais menos para os registos R1, R2 e R9. Esta desigualdade nestes registos deve-se à diferença na biblioteca do eCos™ para implementar a função em *software* e em *hardware*, pois em *software* a função ocupa mais posições na memória do que em *hardware*, fazendo com que os endereços guardados nestes registos sejam diferentes.

Na Figura 50 observa-se a ocorrência de interrupção quando Hrtos está a operar, repara-se que caso está situação acontece (pouco provável mas não impossível) Hrtos finalizará a sua execução primeiro, e devolverá o controlo ao CPU que executará a ISR respetiva. Este acontecimento pode, caso aconteça, aumentar o *overhead* do atendimento da interrupção num máximo de 3.045μs.

B. Análise de Performance temporal

A nível da execução das funções 'load_context' e 'switch_context' examinando o gráfico da Figura 51 verifica-se que as funções realizadas por Hrtos são finalizadas em metade do tempo do que eram finalizadas por *software*, resultando num aumento de *performance* de 50%. Este aumento de performance só pode ser notado caso se utiliza um programa com múltiplas *threads* onde existe preempção múltiplas vezes.

Os resultados apresentados na Figura 52 confirmam esta ultima afirmação. Para a simulação A, só existiu um 'load_context' e uma única comutação de contexto pois as duas *threads* tinham

prioridades diferentes e não existiam mecanismo de sincronização que possibilitavam preempção, levando a execução total de uma *thread* e no seu final o lançamento da segunda *thread*. Para a simulação B, apesar das *threads* possuirem o mesmo grau de prioridades, o escalonador tinha a capacidade de realizar *timeslice*, o que possibilitou um total de 28 comutações de contexto até uma das *threads* finalizar.

C. Análise das Unidades lógicas consumidas

Olhando para a Figura 54, verifica-se que a adição do Hrtos e das suas interfaces acrescentou ao projeto um total de 1052 LUTs e 217 Flip-flops. Este número, comparado com a totalidade de recursos que possuímos na Virtex 5, aumentou a ocupação do projeto de aproximadamente 2%.

Examinado a Figura 55 e a Figura 56, chega-se a conclusão que das 1052 LUTs adicionadas ao projeto só 480 provêm do interior do módulo Hrtos, a nível percentual isto traduz-se por uma utilização inferior a 1% dos recursos da placa por parte de Hrtos. O resto (cerca de 1.2% adicional) deve-se aos *wires* de ligação e controlo aos módulos externos (*register file*, *spr* e ligação ao barramento Wishbone) que consomem um grande número de *slice.*

Destes 480 LUTS e 212 Flip-flop utilizados internamente por Hrtos, verifica-se que as funções em *hardware* 'load_context' e 'switch_context' consomem mais de metade destes recursos, num total de 56%, o que é razoável tendo em conta as operações que realizam.

D. Análise da energia dissipada

A potência consumida está dividida em duas parcelas, a potência estática, que é a potência utilizada pelas unidades lógicas usadas nos módulos, devido a correntes de fuga; e a potência dinâmica que advém da utilização/comutação dos valores destas unidades.

Os resultados apresentados na Figura 58 demonstram que à energia dissipada, nas duas simulações, por OR1200 com Hrtos é superior a energia dissipada por OR1200 sem Hrtos. No entanto esta diferença de consumo é mínima (aproximadamente 0,5mJ) e não chega a ser um entrave à adição de novos módulos.

Capítulo 6

Conclusões

6.1 Conclusões

Nesta dissertação foi apresentado a migração de funcionalidades do sistema operativo eCos™ para *gateware* do processador OR1200. Para tal foi necessário realizar um estudo da composição e do funcionamento do sistema operativo de tempo real e do processador. Com base nesse estudo efetuou-se a migração da operação de comutação de contexto.

As migrações efetuadas, resultaram na criação de um novo módulo de *hardware*, o Hrtos, que está *tightly-coupled* ao processador, partilhando o mesmo *register file* e *special purpose registers*. As operações que este realiza efetuam as mesmas tarefas do que as tarefas realizadas em *software* por parte do CPU. O método de implementação permite ao utilizador ativar ou desativar certas funcionalidades de Hrtos, recorrendo a Macros, quando se está a compilar as bibliotecas do eCos™ e a sintetizar o SOC para a FPGA.

Hrtos possui uma ligação *loosely-coupled* à memória de dados, recorrendo ao barramento externo Wishbone. Esta abordagem foi realizada de modo a poder utilizar Hrtos com a versão base do processador. No entanto, também estava destinado realizar uma ligação a cache de dados, de modo a melhorar o desempenho do acesso aos dados, mas um problema com o suporte do eCos™ para as caches impossibilitou esta implementação.

A nível dos testes realizados, os resultados obtidos são promissores. Estes revelam um aumento na performance temporal, diminuindo para a metade o *overhead* na comutação de contexto. Esta diminuição de tempo deve-se ao retiro de ações supérfluas que o processador realiza e que Hrtos não necessita de efetuar. Este aumento é mais notável quando é executado um programa com múltiplas *threads* onde o escalonador efetua múltiplas preempções e/ou *timeslice*. Os resultados das unidades lógicas usadas divulgam um aumento de 2% de ocupação de área na placa, no entanto, a maior parte deste acréscimo deve-se à adição dos interfaces com os módulos existentes e não diretamente com as funções migradas, prevendo-se um uso mínimo de *slices* para as próximas migrações. A nível de dissipação de energia, regista-se um aumento que não chega a ser preocupante. Para uma implementação onde existe muito ou pouco processamento por parte de Hrtos a dissipação de energia está na ordem dos 50 mJ. Maior parte deste valor deve-se a correntes de fuga (determinadas a partir da potência estática) e não propriamente da utilização/comutação das *slice* (determinada com base na potência dinâmica).

Com base no trabalho efetuado e nos resultados obtidos pode-se afirmar que o desempenho do SOC melhorou para um aumento mínimo de potência e unidades lógicas consumidas, tornando este tipo de implementação viável para sistemas *multithreading*.

6.2 Trabalho Futuro

O projeto atual encontra-se a funcionar cumprindo os requisitos propostos, no entanto este pode ser melhorado, tornando-o mais completo e aumentando o seu desempenho, sendo proposto como trabalho futuro os seguintes pontos:

- Migrar mais funcionalidades, isto é, efetuar mais funções em *hardware* de modo a poder ter um maior grau de comparação entre as métricas propostas.
- Efetuar uma interface com as caches, sendo necessário primeiro corrigir erros no suporte a este *hardware* nas bibliotecas do eCos™.
- Ligar Hrtos ao controlador de interrupções, de modo a conseguir remover a latência já existente e originada por Hrtos.

- Adicionar as macros de configuração a GUI Configtool, de modo a selecionar ou desseleccionar as funções operadas por Hrtos, recorrendo a esta ferramenta simples e fácil de utilizar por parte do utilizador final.

Referências

[1] Arnaldo S.R. Oliveira, "The ARPA-MT Embedded SMT Processor and its RTOS hardware acelerator," *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS*, vol. 58, no. 3, pp. 890-904, Março 2011, ISBN: 0278-0046.

[2] Nichil Gupta, "A hardware Scheduler for Real-Time Multiprocessor System on Chip," *23rd International Conference on VLSI Design*, pp. 264-269, 2010, ISBN: 978-1-4244-5541-6.

[3] Susanna Nordstorm, "Application Specific Real-Time Microkernel in hardware," *14th IEEE-NPSS Real Time Conference*, pp. 333-336, 2005, ISBN: 7803-9183-7.

[4] Terance P. Wijesenghe, "Design and Implementation of a Multithreaded Softcore Processor with Tightly Coupled Real-Time Operating System," College of Engineering and Mineral Resources a WVU, Morgantown, West Virginia, Dissertação (Mestrado em Engenharia Eletrónica) 2008.

[5] Xilinx inc., ML505/ML506/ML507 Evaluation Platform User Guide, Maio 2011.

[6] R. Inam, "Support for Hierarchical Scheduling in FreeRTOS," *technical program at IEEE ETFA*, pp. 1-10, Setembro 2011.

[7] Jung-Guk Kim and Mon Hae Kim, "TMO-eCos: An eCos-based Real-Time Micro Operating System Supporting Execution of a TMO structured Program," *the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pp. 182-189, Junho 2005, ISBN:0-7695-2356-0.

[8] Giovanni Di Sirio. (2011, Outubro) ChibiOS/RT free embedded RTOS. [Online]. http://chibios.sourceforge.net/html/pages.html

[9] (2011, Dezembro) Aeroflex Gaisler. [Online]. http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=156&Itemid=104

[10] S. Saez, A. Crespo, and A. Garcia, "A hardware scheduler for complex real-time systems," *Proceedings of the IEEE International Symposium on Industrial Electronics*, pp. 43-48, 1999, ISBN: 0-7803-5662-4.

[11] Naotaka Maruyama, Tohru Ishihara, and Hiroto Yasuura, "An RTOS in Hardware for Energy Efficient Software-based TCP/IP Processing," *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, pp. 58-63, 2010; ISBN: 978-1-4244-7954-2.

[12] Anthony J. Massa, *Embedded Software Development With eCos*. New Jersey, U.S.A: Prentice Hall Professional Technical Reference, 2003.

[13] Julius Baxter, Michael Unneback, and Marcus Erlandson. (2012, janeiro) OpenCore.org. [Online]. http://opencores.org/openrisc,or1200

[14] Damjan Lampret, "OpenRISC 1200 IP Core Specification," OpenCores, 2001.

[15] OpenCores.org, "WHISBONE SoC Interconnection Architecture for Portable IP Cores," Silicore, 2002.

[16] Julius Baxter, "ORPSoC User Guide," OpenCores, 2011.

[17] Jacob Gorban, "UART IP Core Specifications," OpenCores, 2012.

Bibliografia

1. Anthony J. Massa, *Embedded Software Development With eCos*. New Jersey, U.S.A: Prentice Hall Professional Technical Reference, 2003.

2. Julius Baxter, "Open Source Hardware Development and the OpenRISC Project," IMIT, Tese de Mestrado (Computer Science and Communication) 2011.

3. OPENCORES.org, "WISHBONE SoC Interconnection Architecture for Portable IP Cores," Silicore, 2002.

4. Damjan Lampret, "OpenRISC 1200 IP Core Specification," OpenCores, 2001.

5. Damjan Lampret, Chen-Min Chen, and Marko et Al Milnar, "OpenRISC 1000 Architecture Manual," OpenCores, 2006.

6. Jeremy Bennett and Julius Baxter, "OpenRISC 1200 Supplementary Programmer's Reference Manual," OpenCores, 2010.

7. Julius Baxter, "ORPSoC User Guide," OpenCores, 2011.

ANEXOS

Anexo I – Instalação da GNU-*Toolchain*

Usando um computador com Linux instalado ou através de uma máquina virtual que corra o Linux, abrir um terminal e efetuar o seguinte comando para adquirir os pré-requisitos de instalação:

- *$ sudo apt-get -y install build-essential make gcc g++ flex bison patch texinfo libncurses5-dev libmpfr-dev libgmp3-dev libmpc-dev libzip-dev python-dev libexpat1-dev*

Após a instalação destas ferramentas efetuar o *download* da *toolchain* que está disponível através de um repositório SVN. Usando o terminal efetuar o comando:

- *$ svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/gnu-src*

Assim que o *download* terminar, dirigir-se para o interior da pasta "gnu-src", efetuando:

- *$ cd /onde esta estiver guardada/gnu-src*

No interior desta pasta existem duas *toolchain,* a *newlib toolchain* (a que queremos) que criará as ferramentas com o prefixo 'or32-elf' e a uClibc *toolchain* que criará as ferramentas com o prefixo 'or32-linux'. Usando a script 'bld-all.sh' que se encontra no interior da pasta 'gnu-src', efetuar a compilação da *newlib toolchain* com o seguinte comando:

- *$ ./bld-all.sh –force –prefix /opt/openrisc –or1ksim-dir /opt/or1ksim –no-uclibc –no-or32-linux*

Após efetuar este ultimo comando, todas as ferramentas como compilador, assemblador, *linker* entre outras estarão criadas e prontas a serem utilizadas, no entanto não estarão disponíveis na *path* atual. Para estas ferramentas estarem sempre disponíveis na *path,* editar o ficheiro '*profile'* usando o comando:

- *$ sudo gedit /etc/profile*

Isto abrirá o ficheiro 'profile' com autorização de '*super* utilizador', no final do ficheiro acrescentar a seguinte linha:

- export PATH=/opt/openrisc/bin/:$PATH

Finalmente terão a *newlib toolchain* com as suas ferramentas disponíveis a partir do terminal em qualquer altura.

Anexo II – Obtenção, configuração e compilação de uma aplicação com as bibliotecas do eCos™

Assim que a *toolchain* esteja instalada, poder-se-á efetuar o *download* do repositório do eCos™ e das suas ferramentas. No terminal usando o comando abaixo efetuar-se-á o *download* do repositório:

- *$ svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/rtos/ecos-3.0*

Quando o *download* terminar é necessário obter duas ferramentas que permitirão criar e editar as opções do eCos™ para a arquitetura de processador desejada. Estas ferramentas são:

- ecosconfig
- configtool

A primeira ferramenta, 'ecosconfig', permitirá criar e editar as opções do eCos™, através da linha de comandos. Para utilizar 'ecosconfig' é necessário criá-la, para tal, dirigir-se ao repositório do eCos™, efetuar a configuração e a compilação da ferramenta:

- *$ cd /localização da pasta/ecos-3.0*
- *$ ./configure –prefix=/localização desejada*
- *$ Make*
- *$ Make install*

A segunda ferramenta, 'Configtool', é uma GUI que permitirá ajustar as opções de configurações possíveis de uma forma simples e intuitiva. Para obter a ferramenta efetuar:

- *$ Wget http://www.ecoscentric.com/snapshots/configtool-100305.bz2*
- *$ bunzip2 configtool-100305.bz2*
- *$ chmod u+x configtool-100305*
- *$ ln -s configtool-100305 /usr/local/bin/Configtool*

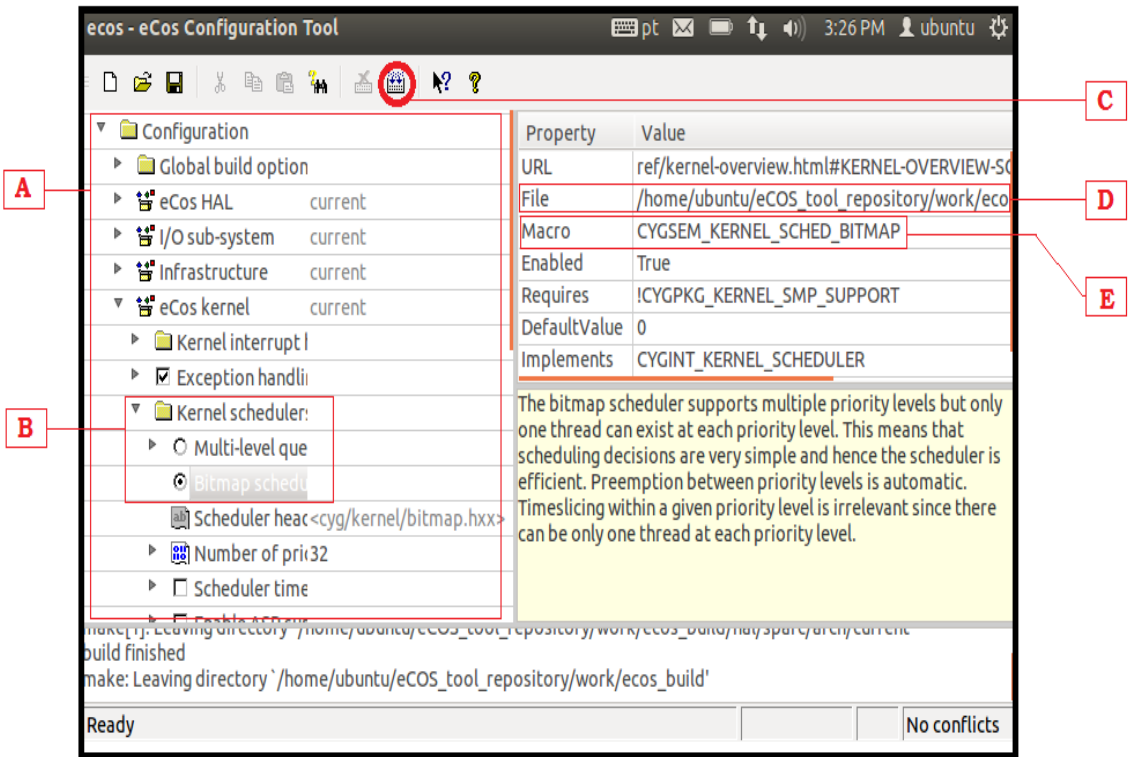
Agora que o *packages* do eCos™ e suas ferramentas estão disponíveis criar-se-á o projeto para a arquitetura OpenRISC. Para tal usando o 'ecosconfig' realizar:

- $ ecosconfig new orpsoc

Este último comando criará um ficheiro com o nome 'ecos.ecc'. Abrir o ficheiro usando 'Configtool' da seguinte forma:

- $ configtool ecos.ecc

Abrir-se-á uma janela como apresentado na Figura 59, onde poderá efetuar-se as seleções das opções desejadas.



Legenda:
A- Opções de configuração do eCOS (kernel, Hal e bibliotecas de compatibilidade)
B- exemplo: efetuar escolha do escalonador
C- Botão para compilar as bibliotecas, previamente escolhidas/configuradas.
D- Informação sobre onde se encontra o ficheiro da implementação da opção selecionada
E- Macro que define a opção selecionada, utilizada na hora de compilar as bibliotecas.

Figura 59: GUI Configtool – Janela principal

Para utilizar Configtool para compilar as librarias é necessário indicar-lhe onde estão as ferramentas de compilação do *host* e do *target*, deslocando-se ao separador /tools/path, Figura 60. Também é necessário indicar-lhe onde se encontram os *packages* necessários, deslocando ao separador /build/repositor, Figura 61.

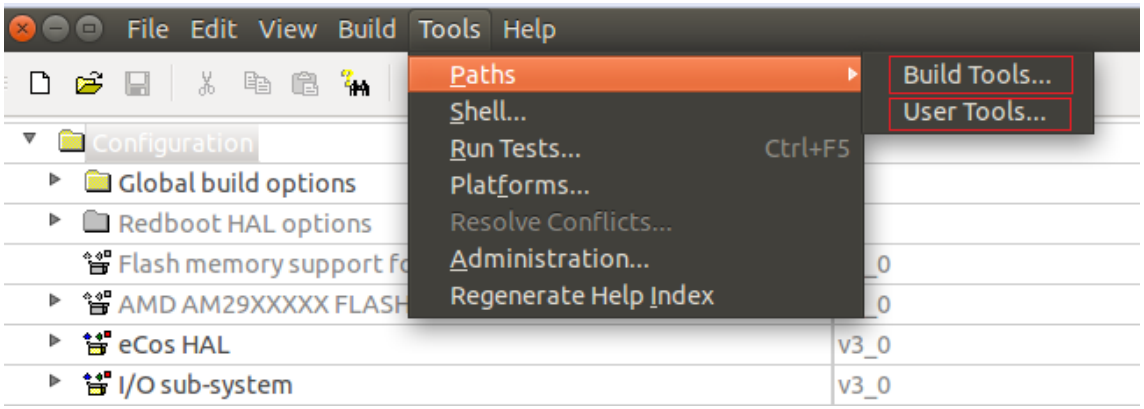


Figura 60: GUI Configtool- localizar ferramentas

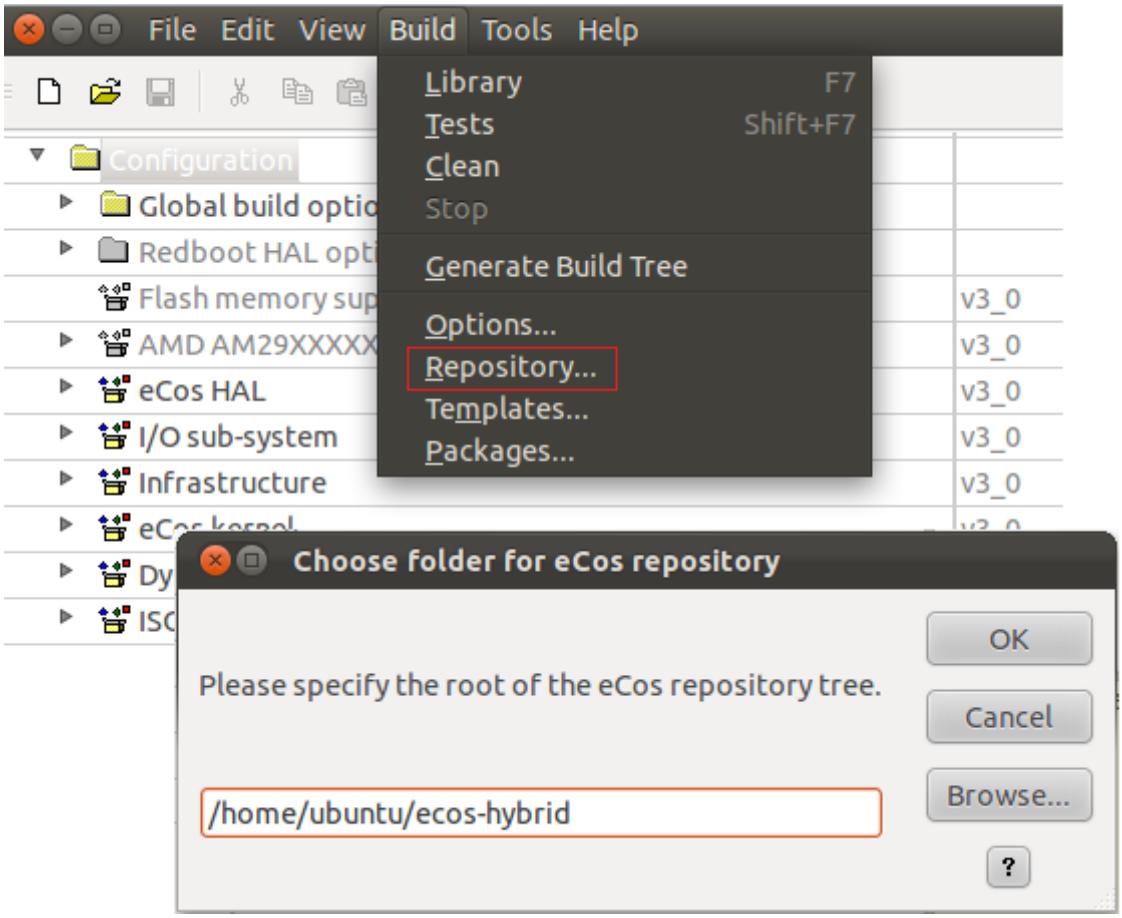


Figura 61: GUI Configtool- localizar repositório

Na hora de escolher as opções desejadas, algumas destas poderão efetuar conflitos com outras opções selecionadas. Para informar o utilizador destas alterações, esta ferramenta abrirá uma janela de resolução de conflitos. A Figura 62 mostra a janela de conflito gerada, quando se escolhe o escalonador 'Mlqueue' em detrimento do escalonador 'bitmap'.

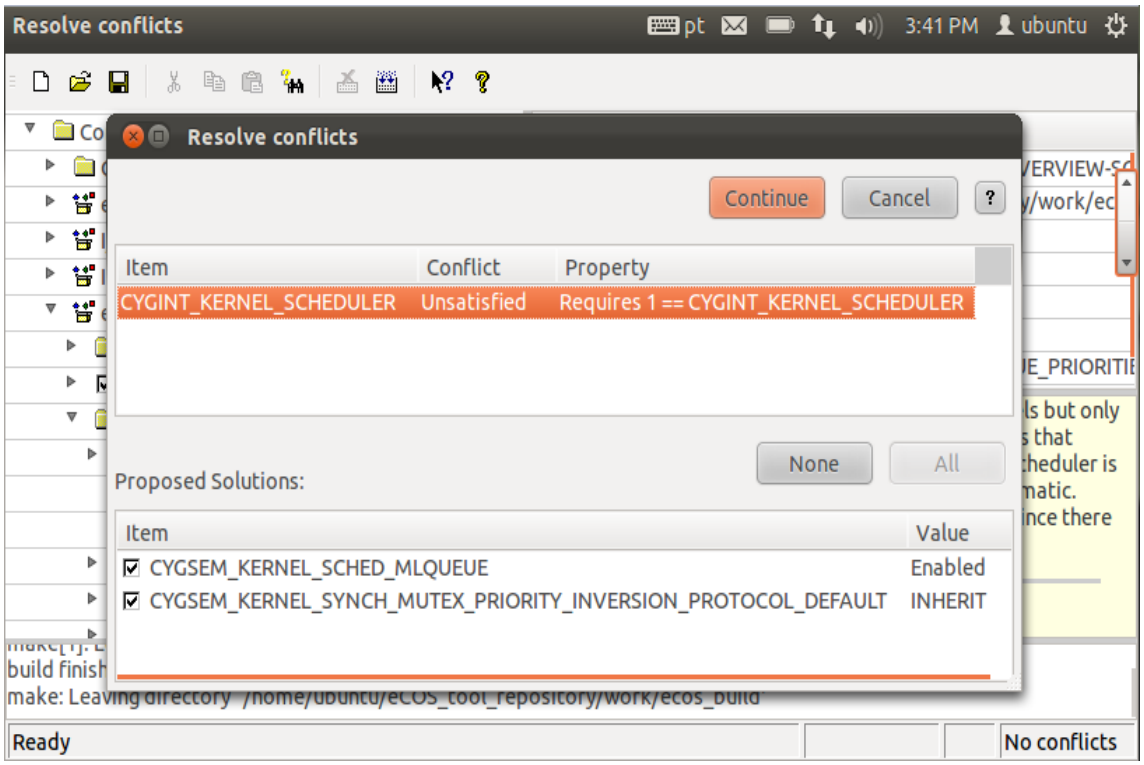


Figura 62: GUI Configtool- Janela de resolução de conflitos

Após escolher as opções desejadas efetuar a compilação das librarias, C na Figura 59, serão criados os ficheiros, 'libtarget.a' e 'targe.ld', localizados na pasta 'ecos_install' na mesma diretoria do que 'ecos.ecc'.

Para efetuar a compilação de uma aplicação 'teste' com as librarias do eCos™ usar os seguintes prefixos:

- *$ or32-elf-gcc -g -O2 -nostdlib*
  -I/local/ecos_build/install/include
  - L/local/ecos_build/install/lib
  -T/local/ecos_build/install/lib/target.ld teste.c -o TESTE

Anexo III – ISA do OR1200 – ORBIS32

INSTRUCTION SET	31 - 24	23 - 16	15 - 8	7 - 0	Implementation Specific	Exception
l.j I	0 0 0 0 0 0 I I	0x II IIII			PC <- EXTs (imediato << 2) + jinsnaddr	none
	0x00 -> 0x03	IMEDIATO 26 bits			--> delay 1 insn	
jump						
l.jal I	0 0 0 0 0 1 I I	0x II IIII			PC <- EXTs (imediato << 2) + jinsnaddr	none
	0x04 -> 0x07	IMEDIATO 26 bits			LR <- DelayINSNaddr + 4	
jump and link					--> delay 1 insn	
l.bnf I	0 0 0 0 1 1 I I	0x II IIII			EA <- EXTs (imediato << 2) + binsnaddr	none
	0x0C -> 0x0F	IMEDIATO 26 bits			PC <- EA Se SR[F]==0	
branch if not flag					--> delay 1 insn	
l.bf I	0 0 0 1 0 0 I I	0x II IIII			EA <- EXTs (imediato << 2) + binsnaddr	none
	0x10 -> 0x13	IMEDIATO 26 bits			PC <- EA Se SR[F]==1	
branch if flag					--> delay 1 insn	
l.nop	0 0 0 1 0 1 1 0		0XIIII			none
	0X15	Reserved	IMEDIATO 16 bits			
l.macrc RD	0 0 0 1 1 0 d d d d d	4 bits I	0XIIII		RD <- MACLO	none
	0x18 -> 0x1B	Registo D Reserved	IMEDIATO 17 bits		MACHI[31:0]MACLO[31:0]<- 0	
MAC Read and Clear						
l.movhi RD, I	0 0 0 1 1 0 d d d d d	4 bits 0	0XIIII		RD <- EXTz (I) << 16	none
	0x18 -> 0x1B	Registo D Reserve 0	IMEDIATO 16 bits			
Move imediate high						
l.sys I	0 0 1 0 0 0 0 0	0 0 0 0	0 0 0 0		system-call-execption (I)	system call
	0X 2000		Imediato 16 bits			
system call						
				0		
l.trap I	0 0 1 0 0 0 0 1	0 0 0 0	0 0 0 0		SE SR[i] ==1 então trap-exception ()	trap execption
	0X 2100		Imediato 16 bits			
Trap						
l.msync	0 0 1 0 0 0 1 0	0 0 0 0	0 0 0 0			none
	0X2200 0000	A execução desta instrução resulta da realização de todos as operações de load/store antes do				
memory syncronization	cores RISC continuar a execução					
l.psync	0 0 1 0 0 0 1 0	1 0 0 0	0 0 0 0			none
	0X2280 0000	A execução desta instrução resulta da realização de todos as instruções fetched ante da l.psync.				
pipeline syncronization	Assim que estiverem todas completas, a instrução fetched depois de l.psync é flushed do pipeline e fetched outra vez.					
l.csync	0 0 1 0 0 0 1 1	0 0 0 0	0 0 0 0			none
	0x2300 0000					
context syncronization	os RISC cores recomeçam com um pipeline de instrução vazias e um novo contexto para todas as unidades (ex: MMU)					
l.jr RB	0 1 0 0 0 1	10 bits	b b b b b	11 bits	PC <- RB	none
	0x44 -> 0x47	reserved	Registo B	Reserved		
jump register					--> delay 1 insn	
l.jalr RB	0 1 0 0 1 0	10 bits	b b b b b	11 bits	PC <- RB	none
	0x48 -> 0x4B	reserved	Registo B	Reserved	LR <- DelayINSNaddr + 4	
jump and link register					--> delay 1 insn	
l.maci RA, I	0 1 0 0 1 1 I I	I I I 5 bits	b b b b b	11 bits	temp[31:0] <- RA * EXTs(I)	none
multiply immediate	0x4C -> 0x4F	Imediato 5MSI Reserved	Registo B	Imediato 11 LSB	MACHI[31:0], MACLO[31:0]<- temp[31:0] +	
signed and acumulate					MACHI[31:0]MACLO[31:0]	
l.lwz Rd, I (RA)	1 0 0 0 0 1 d d	d d d a a a a	0XIIII		EA <- EXTs (imediato) + RA	TLB miss Alignment
load word and	0x84 -> 0x87	Registo D Registo A	IMEDIATO 16 bits		RD[31:0] <- (EA) [31:0]	Page fault
extend with zero						Bus Error
l.lws Rd, I (RA)	1 0 0 0 1 0 d d	d d d a a a a	0XIIII		EA <- EXTs (imediato) + RA	TLB miss Alignment
load word and	0x88 -> 0x8B	Registo D Registo A	IMEDIATO 16 bits		RD[31:0] <- (EA) [31:0]	Page fault
extend with sign						Bus Error
l.lbz Rd, I (RA)	1 0 0 0 1 1 d d	d d d a a a a	0XIIII		EA <- EXTz (imediato) + RA	TLB miss
load Byte and	0x8C -> 0x8F	Registo D Registo A	IMEDIATO 16 bits		RD[7:0] <- (EA) [7:0]	Page fault
extend with zero					RD[31:8] <-0	Bus Error
l.lbs Rd, I (RA)	1 0 0 1 0 0 d d	d d d a a a a	0XIIII		EA <- EXTs (imediato) + RA	TLB miss
load Byte and	0x90 -> 0x93	Registo D Registo A	IMEDIATO 16 bits		RD[7:0] <- (EA) [7:0]	Page fault
extend with sign					RD[31:8] <-(EA) [7]	Bus Error
l.lhz Rd, I (RA)	1 0 0 1 0 1 d d	d d d a a a a	0XIIII		EA <- EXTs (imediato) + RA	TLB miss Alignment
load half word	0x94 -> 0x97	Registo D Registo A	IMEDIATO 16 bits		RD[15:0] <- (EA) [15:0]	Page fault
and extend with zero					RD[31:16] <-0	Bus Error

INSTRUCTION SET	31 - 24	23 - 16	15 - 8	7 - 0	Implementation Specific	Exception
l.lhs Rd, I (RA)	1 0 0 1 1 0 d d d d a a a a	0XIIII			EA <- EXTs (imediato) + RA	TLB miss Alignment
load half word	0x98 -> 0x9B Registo D	Registo A	IMEDIATO 16 bits		RD[15:0] <- (EA) [15:0]	Page fault
and extend with sign					RD[31:16] <-(EA) [15]	Bus Error
l.addi Rd,Ra,I	1 0 0 1 1 1 d d d d a a a a	0XIIII			RD <- RA + EXTs (imediato)	range exception
	0x9C -> 0x9F Registo D	Registo A	IMEDIATO 16 bits		SR [CY] <- carry	
					SR[OV] <-overflow	
l.addic Rd,Ra,I	1 0 1 0 0 0 d d d d a a a a	0XIIII			RD <- RA + EXTs (imediato) +SR[CY]	range exception
	0xA0 -> 0xA3 Registo D	Registo A	IMEDIATO 16 bits		SR [CY] <- carry	
					SR[OV] <-overflow	
l.andi Rd,Ra,I	1 0 1 0 0 1 d d d d a a a a	0XIIII			RD <- RA + EXTz (imediato)	none
	0xA4 -> 0xA7 Registo D	Registo A	IMEDIATO 16 bits			
l.ori Rd,Ra,I	1 0 1 0 1 0 d d d d a a a a	0XIIII			RD <- RA | EXTz (i)	none
	0xA8 -> 0xAB Registo D	Registo A	IMEDIATO 16 bits			
l.xori Rd,Ra,I	1 0 1 0 1 1 d d d d a a a a	0XIIII			RD <- RA XOR EXTs (i)	none
exclusive Or with	0xAC -> 0xAF Registo D	Registo A	IMEDIATO 16 bits			
imm half word						
l.muli Rd,Ra,I	1 0 1 1 0 0 d d d d a a a a	0XIIII			RD <- RA * I	range exception
multiply	0xB0 -> 0xB3 Registo D	Registo A	IMEDIATO 16 bits		SR [CY] <- carry	
imediate signed					SR[OV] <-overflow	
l.mfspr Rd, RA ,I	1 0 1 1 0 1 d d d d a a a a	0XIIII			RD <- spr (RA | I)	none
move from special	0xB4 -> 0xB7 Registo D	Registo A	IMEDIATO 16 bits			
purpose register						
l.slli Rd,Ra, I	1 0 1 1 1 0 d d d d a a a a			0 0	RD [31:I] <-RA [31-I:0]	none
shift left logical	0xB8 -> 0xBB Registo D	Registo A	Reserved 8 bits	OP Imedi 6 bits	RD[I-1:0] <- 0	
with imediate						
l.srli Rd,Ra, I	1 0 1 1 1 0 d d d d a a a a			0 1	RD [31-I:0] <-RA [31:I]	none
shift right logical	0xB8 -> 0xBB Registo D	Registo A	Reserved 8 bits	OP Imedi 6 bits	RD[31:32-I] <- 0	
with imediate						
l.srai Rd,Ra, I	1 0 1 1 1 0 d d d d a a a a			1 0	RD [31-I:0] <-RA [31:i]	none
shift right arithmeti	0xB8 -> 0xBB Registo D	Registo A	Reserved 8 bits	OP Imedi 6 bits	RD[31:32-1] <- Ra[31]	
c with imediate						
l.rori Rd,Ra, I	1 0 1 1 1 0 d d d d a a a a			1 1	RD[31- I: 0] <- RA[31:I]	none
rotate right	0xB8 -> 0xBB Registo D	Registo A	Reserved 8 bits	OP Imedi 6 bits	RD[31:32-I] <- RA[I-1:0]	
with imediate						
l.sfeqi RA, I	1 0 1 1 1 1 0 0 0 0 a a a a	0XIIII			SR [F] <- RA == EXTs (I)	none
Set Flag if	0xBC0 -> 0xBC1	Registo A	IMEDIATO 16 bits			
equal immediate						
l.sfnei RA, I	1 0 1 1 1 1 0 0 0 1 a a a a	0XIIII			SR [F] <- RA != EXTs (I)	none
Set Flag if Not	0xBC2 -> 0xBC3	Registo A	IMEDIATO 16 bits			
Equal Immediate						
l.sfgtui RA, I	1 0 1 1 1 1 0 0 0 1 0 a a a a	0XIIII			SR [F] <- RA > EXTs (I)	none
Set Flag if >	0xBC4 -> 0xBC5	Registo A	IMEDIATO 16 bits			
than I unsigned						
l.sfgeui RA, I	1 0 1 1 1 1 0 0 0 1 1 a a a a	0XIIII			SR [F] <- RA >= EXTs (I)	none
Set Flag if	0xBC6 -> 0xBC7	Registo A	IMEDIATO 16 bits			
>= I unsigned						
l.sfltui RA, I	1 0 1 1 1 1 0 0 1 0 0 a a a a	0XIIII			SR [F] <- RA < EXTs (I)	none
Set Flag if	0xBC8 -> 0xBC9	Registo A	IMEDIATO 16 bits			
< than I unsigned						
l.sfleui RA, I	1 0 1 1 1 1 0 0 1 0 1 a a a a	0XIIII			SR [F] <- RA <= EXTs (I)	none
Set Flag if	0xBCA -> 0xBCB	Registo A	IMEDIATO 16 bits			
<= I unsigned						
l.sfgtsi RA, I	1 0 1 1 1 1 0 1 0 1 0 a a a a	0XIIII			SR [F] <- RA > EXTs (I)	none
Set Flag if	0xBD4 -> 0xBD5	Registo A	IMEDIATO 16 bits			
> I signed						
l.sfgesi RA, I	1 0 1 1 1 1 0 1 0 1 1 a a a a	0XIIII			SR [F] <- RA >= EXTs (I)	none
	0xBD6 -> 0xBD7	Registo A	IMEDIATO 16 bits			
Set Flag if >= I						
l.sfltsi RA, I	1 0 1 1 1 1 0 1 1 0 0 a a a a	0XIIII			SR [F] <- RA < EXTs (I)	none
Set Flag if	0xBD8 -> 0xBD9	Registo A	IMEDIATO 16 bits			
< I signed						
l.sflesi RA, I	1 0 1 1 1 1 0 1 1 0 1 a a a a	0XIIII			SR [F] <- RA <= EXTs (I)	none
	0xBDA -> 0xBDB	Registo A	IMEDIATO 16 bits			
Set Flag if <= I						

INSTRUCTION SET	31 - 24	23 - 16	15 - 8	7 - 0	Implementation Specific	Exception
l.mtspr RA, RB, I	1 1 0 0 0 0 I I I I I a a a a a	b b b b b	11 bits		spr (RA | I) <- RB	none
move to special	0xC0 -> 0xC3	Imediato 5MSI	Registo A	Registo B	Imediato 11 LSB		
purpose register						
l.mac RA, RB	1 1 0 0 0 1	5 bits	a a a a a	b b b b b	7 bits 0 0 0 1	temp[31:0] <- RA *RB	none
multiply signed	0xC4 -> 0xC7	reservec	Registo A	Registo B	Reserved 0X1	MACHI[31:0]MACLO[31:0]<- temp[31:0]	
and acumulate						+ MACHI[31:0]MACLO[31:0]	
l.msb RA, RB	1 1 0 0 0 1		a a a a a	b b b b b	7 bits 0 0 1 0	temp[31:0] <- RA * RB	none
multiply signed	0xC4 -> 0xC7	Reserved	Registo A	Registo B	Reserved 0X2	MACHI[31:0]MACLO[31:0]<-	
and subtract						MACHI[31:0]MACLO[31:0]- temp[31:0]	
l.sw I (RA), RB	1 1 0 1 0 1 I I I I I a a a a a	b b b b b	11 bits		EA <- EXTs(I) + RA	TLB miss Alignment
	0xD4 -> 0xD7	Imediato 5MSI	Registo A	Registo B	Imediato 11 LSB	(EA) [31:0] <- RB [31:0]	Page fault
store word							Bus Error
l.sb I (RA), RB	1 1 0 1 1 0 I I I I I a a a a a	b b b b b	11 bits		EA <- EXTs (imediato) + RA	TLB miss
	0xD8 -> 0xDB	Imediato 5MSI	Registo A	Registo B	Imediato 11 LSB	EA[7:0] <- RB [7:0]	Page fault
store byte							Bus Error
l.sh I (RA), RB	1 1 0 1 1 1 I I I I I a a a a a	b b b b b	11 bits		EA <- EXTs(I) + RA	TLB miss Alignment
	0xDC -> 0xDF	Imediato 5MSI	Registo A	Registo B	Imediato 11 LSB	(EA) [15:0] <- RB [15:0]	Page fault
store half word							Bus Error
l.add Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0	R R R R	0 0 0 0	RD <- RA +RB	range exception
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	x 0 0	0xN	0x0	SR [CY] <- carry	
								SR[OV] <-overflow	
l.addc Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0	R R R R	0 0 0 1	RD <- RA +RB + SR[CY]	range exception
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	x 0 0	0xN	0x1	SR [CY] <- carry	
								SR[OV] <-overflow	
l.sub Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0	R R R R	0 0 1 0	RD <- RA - RB	range exception
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	res 0 0	reserved	0x2	SR [CY] <- carry	
subtract signed								SR[OV] <-overflow	
l.and Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0	R R R R	0 0 1 1	RD <- RA ^ RB	none
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	x 0 0	0xN	0x3		
l.or Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0	R R R R	0 1 0 0	RD[31:0] <- RA |RB	none
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	res 0 0	reserved	0X4		
l.xor Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0	R R R R	0 1 0 1	RD <- RA XOR RB	none
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	res 0 0	reserved	0x5		
exclusive Or									
l.mul Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 1 1	R R R R	0 1 1 0	RD[31:0] <- RA * RB	range exception
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	res 1 1	reserved	0X6	SR [CY] <- carry	
multiply signed								SR[OV] <-overflow	
l.sll Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0 0 0	R R	1 0 0 0	RD [31:RB[4:0]] <-RA [31-Rb[4:0]:0]	none
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	0 0 0 0	reservec	0X8	RD[Rb[4:0]-1:0] <- 0	
shift left logical									
l.srl Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0 0 1	R R	1 0 0 0	RD [31-RB[4:0]: 0] <-RA [31:Rb[4:0]]	none
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	0 0 0 1	reservec	0X8	RD[31:32-Rb[4:0]] <- 0	
shift right logical									
l.sra Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0 1 0	R R	1 0 0 0	RD [31-RB[4:0]: 0] <-RA [31-Rb[4:0]:0]	none
shift right	0xE0 -> 0xE3	Registo D	Registo A	Registo B	0 0 1 0	reservec	0X8	RD[31:32-Rb[4:0]] <- RA[31]	
arithmetic									
l.ror Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 0 0	1 1 R R	1 0 0 0	RD[31- RB[4:0]: 0] <- RA[31:RB]	none
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	res 0 0 1 1		0X8	RD[31:32-RB[4:0]] <- RA[RB[4:0]-1:0]	
rotate right									
l.div Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 1 1	R R R R	1 0 0 1	RD <- RA / RB	range exception
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	res 1 1	reserved	0X9	SR [CY] <- carry se Divisor is 0	
divisão com sinal								SR[OV] <-overflow	
l.divu Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 1 1	R R R R	1 0 1 0	RD <- RA / RB	range exception
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	res 1 1	reserved	0XA	SR [CY] <- carry se Divisor is 0	
divisão sem sinal								SR[OV] <-overflow	
l.mulu Rd,Ra,Rb	1 1 1 0 0 0 d d d d d a a a a a	b b b b b	R 1 1	R R R R	1 0 1 1	RD[31:0] <- RA * RB	range exception
	0xE0 -> 0xE3	Registo D	Registo A	Registo B	res 1 1	reserved	0XB	SR [CY] <- carry	
multiply unsigned								SR[OV] <-overflow	
l.exths Rd,Ra	1 1 1 0 0 0 d d d d d a a a a a	R R R R R	0 0 0 0	R R	1 1 0 0	RD[31:16] <- RA[15]	none
extend half	0xE0 -> 0xE3	Registo D	Registo A	Reserved	0 0 0 0	reservec	0xC	RD[15:0] <- RA [15:0]	
word with sign									

INSTRUCTION SET	31 - 24	23 - 16	15 - 8	7 - 0	Implementation Specific	Exception
l.exthz Rd,Ra	1 1 1 0 0 0 d d d d a a a a	R R R R R R	0 0 1 0 R R	1 1 0 0	RD[31:16] <- 0	none
extend half	0xE0 -> 0xE3	Registo D	Registo A	Reserved	0 0 1 0 reserved	0xC	RD[15:0] <- RA [15:0]	
word with zero						
l.extbz Rd,Ra	1 1 1 0 0 0 d d d d a a a a	R R R R R R	0 0 1 1 R R	1 1 0 0	RD[31:8] <- 0	none
extend Byte	0xE0 -> 0xE3	Registo D	Registo A	Reserved	0 0 1 1 reserved	0xC	RD[7:0] <- RA [7:0]	
with zero						
l.extws Rd,Ra	1 1 1 0 0 0 d d d d a a a a	R R R R R R	0 0 0 0 R R	1 1 0 1	RD <-RA	none
extend word	0xE0 -> 0xE3	Registo D	Registo A	Reserved	0 0 0 0 reserved	0XD		
with sign						
l.extwz Rd,Ra	1 1 1 0 0 0 d d d d a a a a	R R R R R R	0 0 0 1 R R	1 1 0 1	RD <-RA	none
extend word	0xE0 -> 0xE3	Registo D	Registo A	Reserved	0 0 0 1 reserved	0XD		
with zero						
l.cmov Rd,Ra,Rb	1 1 1 0 0 0 d d d d a a a a	b b b b R 0 0	R R R R	1 1 1 0	RD <- SR [F] ? RA : RB	none
	0xE0 -> 0xE3	Registo D	Registo A	Registo B res 0 0	0xN	0xE		
conditional move						
l.ff1 Rd,Ra,Rb	1 1 1 0 0 0 d d d d a a a a	b b b b R 0 0	R R R R	1 1 1 1	RD[31:0] <- RA[0]? 1: RA[1]? 2: RA[2]?3:...	none
	0xE0 -> 0xE3	Registo D	Registo A	Registo B res 0 0	reserved	0XF	RA[31]? 32: 0	
find first 1						
l.fl1 Rd,Ra,Rb	1 1 1 0 0 0 d d d d a a a a	b b b b R 0 1	R R R R	1 1 1 1	RD[31:0] <- RA[31]? 32: RA[30]? 31: ...	none
	0xE0 -> 0xE3	Registo D	Registo A	Registo B res 0 1	reserved	0XF	RA[0]? 1: 0	
find last 1						
l.sfeq RA, RB	1 1 1 0 0 1 0 0 0 0 0 a a a a	b b b b b	11 bits		SR [F] <- RA == RB	none
	0xE40 -> 0xE41	Registo A	Registo B	Reserved		
Set Flag if equal						
l.sfne RA, RB	1 1 1 0 0 1 0 0 0 0 1 a a a a	b b b b b	11 bits		SR [F] <- RA != RB	none
Set Flag if	0xE42 -> 0xE43	Registo A	Registo B	Reserved		
Not Equal						
l.sfgtu RA, RB	1 1 1 0 0 1 0 0 0 1 0 a a a a	b b b b b	11 bits		SR [F] <- RA > RB	none
Set Flag if >	0xE44 -> 0xE45	Registo A	Registo B	Reserved		
than unsigned						
l.sfgeu RA, RB	1 1 1 0 0 1 0 0 0 1 1 a a a a	b b b b b	11 bits		SR [F] <- RA >= RB	none
Set Flag if	0xE46 -> 0xE47	Registo A	Registo B	Reserved		
>= than unsigned						
l.sfltu RA, RB	1 1 1 0 0 1 0 0 1 0 0 a a a a	b b b b b	11 bits		SR [F] <- RA < RB	none
Set Flag if	0xE48 -> 0xE49	Registo A	Registo B	Reserved		
< than unsigned						
l.sfleu RA, RB	1 1 1 0 0 1 0 0 1 0 1 a a a a	b b b b b	11 bits		SR [F] <- RA <= RB	none
Set Flag if	0xE4A -> 0xE4B	Registo A	Registo B	Reserved		
<= than unsigned						
l.sfgts RA, RB	1 1 1 0 0 1 0 1 0 1 0 a a a a	b b b b b	11 bits		SR [F] <- RA > RB	none
Set Flag if	0xE54 -> 0xE55	Registo A	Registo B	Reserved		
> than signed						
l.sfges RA, RB	1 1 1 0 0 1 0 1 0 1 1 a a a a	b b b b b	11 bits		SR [F] <- RA >= RB	none
Set Flag	0xE56 -> 0xE57	Registo A	Registo B	Reserved		
if >= than						
l.sflts RA, RB	1 1 1 0 0 1 0 1 1 0 0 a a a a	b b b b b	11 bits		SR [F] <- RA < RB	none
Set Flag	0xE58 -> 0xE59	Registo A	Registo B	Reserved		
if < than signed						
l.sfles RA, RB	1 1 1 0 0 1 0 1 1 0 1 a a a a	b b b b b	11 bits		SR [F] <- RA <= RB	none
	0xE5A -> 0xE5B	Registo A	Registo B	Reserved		
Set Flag if <= than						

Legenda:

Acrónimo	Definição	Acrónimo	Definição	Acrónimo	Definição
a	Registo a	LR	*Link register*	EXTs	*sign-extended*
b	Registo b	SR	*Superivision Register*	EXTs	*sign-extended*
d	Registo d	OV	*Overflow bit*	F	*Flag bit*
R	*reserved*	binsnaddr	*branch instruction address*	PC	*Program Counter*
I	Imediato	EXTZ	*zero-extended*	EA	*Effective Address*
CY	Bit de carry				

Anexo IV – Geração do ficheiro '.coe'

Após a aplicação ser compilada é necessário criar o ficheiro com o código máquina de modo a ser carregado na memória do processador. A *toolchain* possui duas ferramentas que geram os ficheiros '.bin' e '.vmem' usando os comandos:

- $ or32-elf-objcopy –O binary TESTE teste.bin
- $ bin2vmem teste.bin teste.vmem

Visto que a memória usada no projeto é uma primitiva da XILINX que não aceita os dois formatos anteriormente mencionados ('.bin' e '.vmem') é necessário traduzir o ficheiro '.vmem' para '.coe'. Para tal usa-se a ferramenta 'vmem2coe' desenvolvida na realização deste projeto, cujo código está presente na Figura 63.

```c
#include <stdio.h>

int main (int argc,char *argv[])
{
        FILE *vmem;v
        FILE *coe;

        int pos,a,b,c,d;


        vmem=fopen(argv[1],"rb");
        if (vmem==NULL)
        {
                printf("Error opening bin file...\n");
                return(2);
        }
        coe=fopen(argv[2],"wt");
        if (coe==NULL)
        {
                printf("Error opening coe file...\n");
                return(2);
        }

        fprintf(coe,"memory_initialization_radix=16;\n");
        fprintf(coe,"memory_initialization_vector=\n");

        while (fscanf(vmem, "@%X %X %X %X %X\n", &pos, &a, &b, &c, &d)!=EOF)
                fprintf(coe, "%X,\n%X,\n%X,\n%X,\n", a, b, c, d);
        fprintf(coe,"0;");
        fclose (vmem);
        fclose (coe);
 return 0;
 }
```

Ficheiro resultante '.coe'

```
memory_initialization_radix=16;
memory_initialization_vector=
15000000,
15000000,
15000000,
0,
 ...
0;
```

Figura 63: Código da ferramenta 'vmem2coe'

Anexo V – Passagem do projeto para a placa XC5VLX110T.

Após abrir e sintetizar o projeto é necessário mapear os pinos de entrada e saída na placa. Para tal usa-se a ferramenta 'PlanAhead'. Esta encontra-se no separador Tools/PlanAhead/ do ISE da Xilinx, como está apresentado na Figura 64.

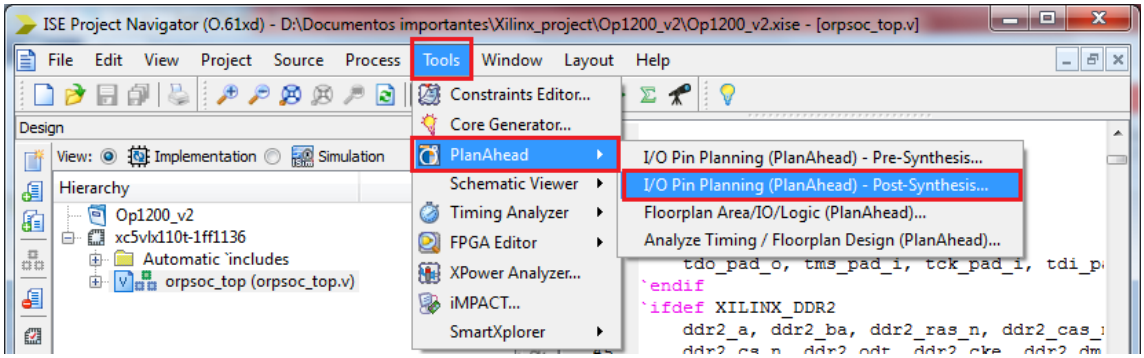


Figura 64: Localização da ferramenta I/O Pin Planning

Será então aberta uma janela, Figura 65, onde se deve mapear os seguintes pinos 'rst_n_pad_i', 'sys_clk_in_p', 'uart0_srx_pad_i' e 'uart0_stx_pad_o'.

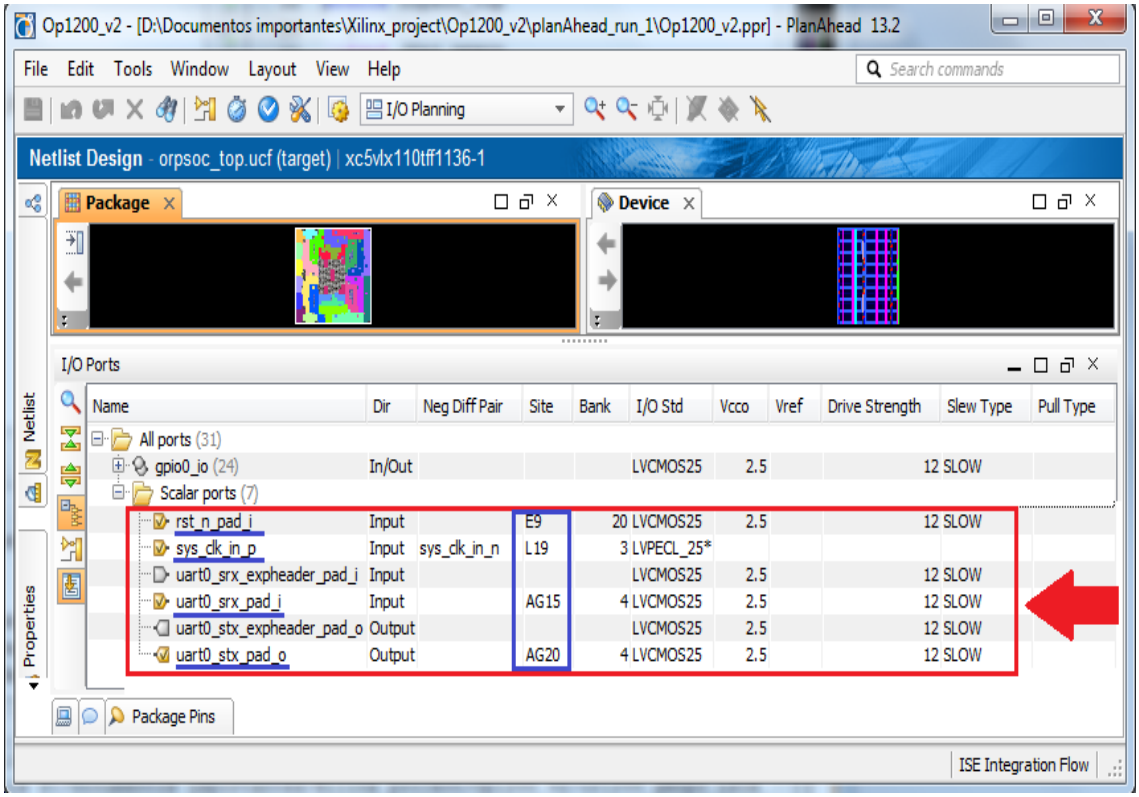

Op1200_v2 - [D:\Documentos importantes\Xilinx_project\Op1200_v2\planAhead_run_1\Op1200_v2.ppr] - PlanAhead 13.2
File Edit Tools Window Layout View Help
Search commands
I/O Planning
Netlist Design - orpsoc_top.ucf (target) | xc5vlx110tff1136-1
Package
Device
I/O Ports
Name Dir Neg Diff Pair Site Bank I/O Std Vcco Vref Drive Strength Slew Type Pull Type
All ports (31)
gpio0_io (24) In/Out LVCMOS25 2.5 12 SLOW
Scalar ports (7)
rst_n_pad_i Input E9 20 LVCMOS25 2.5 12 SLOW
sys_clk_in_p Input sys_clk_in_n L19 3 LVPECL_25* 12 SLOW
uart0_srx_expheader_pad_i Input LVCMOS25 2.5 12 SLOW
uart0_srx_pad_i Input AG15 4 LVCMOS25 2.5 12 SLOW
uart0_stx_expheader_pad_o Output LVCMOS25 2.5 12 SLOW
uart0_stx_pad_o Output AG20 4 LVCMOS25 2.5 12 SLOW
Package Pins
ISE Integration Flow
Netlist
Properties

Figura 65: Ferramenta 'PlanAhead'

Assim que os pinos estiverem mapeados, abrir a ferramenta 'iMPACT', esta encontra-se no separador /tools/iMPACT, Figura 66.

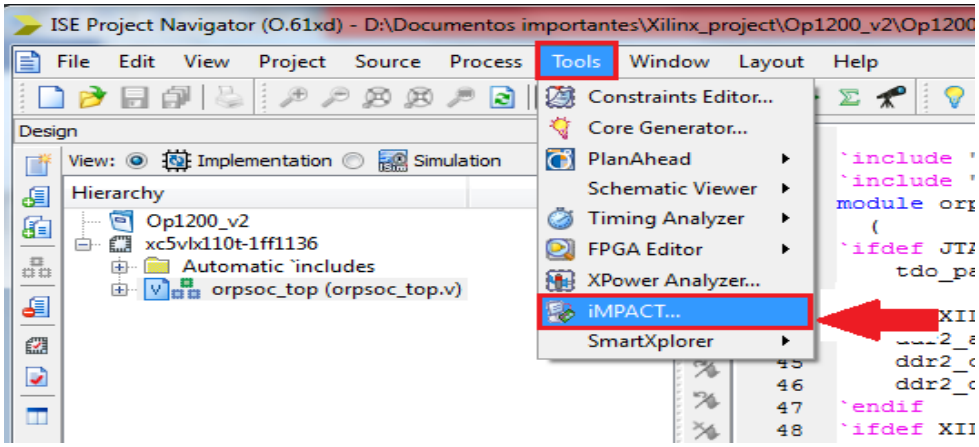


Figura 66: Localização da ferramenta 'iMPACT'.

Neste momento será necessário preparar a placa, sendo assim ligar o USB/JTAG a placa (pinos PC4) e ao computador, assim como o conversor USB/RS232 com inversor a COM1. Para efetuar o carregamento do projeto, para a memória volátil, colocar os pinos SW3.6 e SW3.8 a 1 e os restantes a zero, Figura 67.

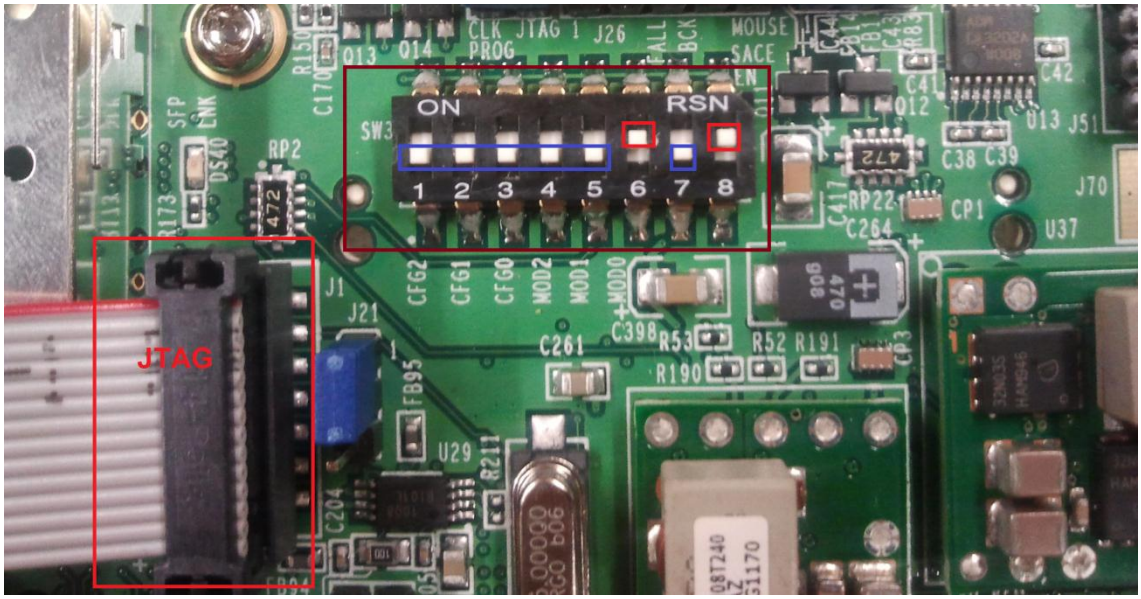


Figura 67: Configuração dos pinos SW3 da placa XC5VLX110T e ligação JTAG.

Após preparar a placa voltar ao iMPACT e efetuar 'Boundary Scan', de seguida na janela branca efetuar clique direito com o rato e efetuar 'Initialize Chain', Figura 68.

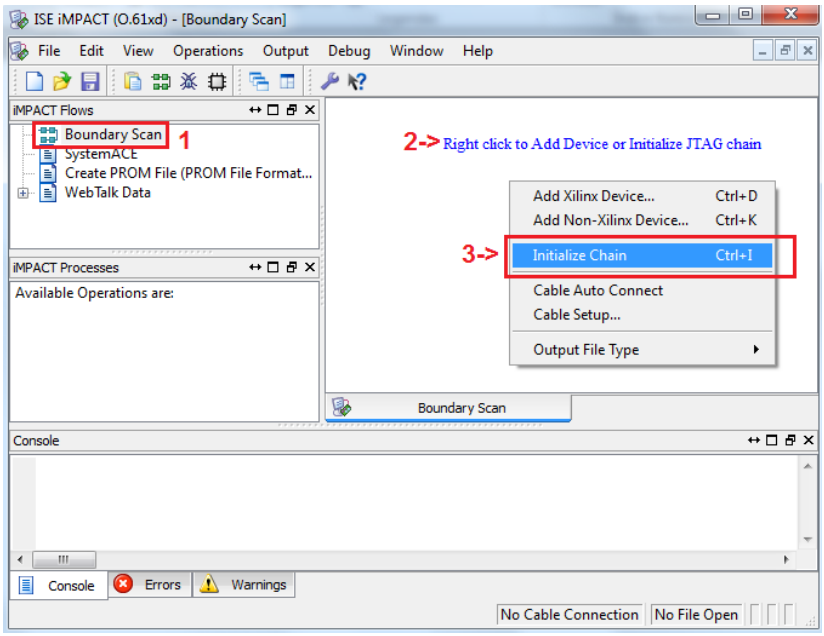


Figura 68: iMPACT- Inicializar

Esta última ação desencadeará a abertura de uma janela para decidir quais são os ficheiros de configuração a carregar, Figura 69, efetuar 'bypass' de todos os passos até aparecer o ficheiro '.bit', seleciona-se o ficheiro e efetua-se 'open'.

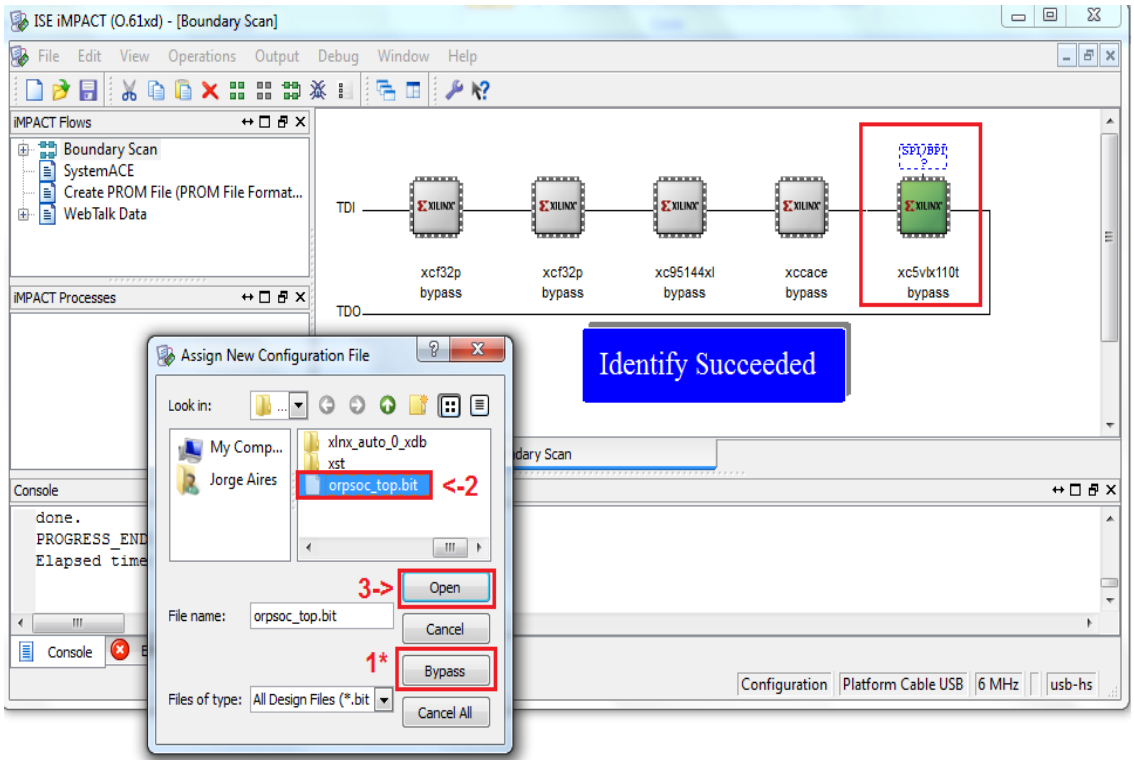


Figura 69: iMPACT- Selecionar ficheiro do 'Design'

Assim que carregar em 'Open' abrir-se-á uma janela a pedir para adicionar um ficheiro 'PROM', selecionar 'não' e efetuar cancelar na próxima janela que abrir. De seguida selecionar 'XC5VLX110T' com o ficheiro '.bit' já inicializado e efetuar 'Program', Figura 70.

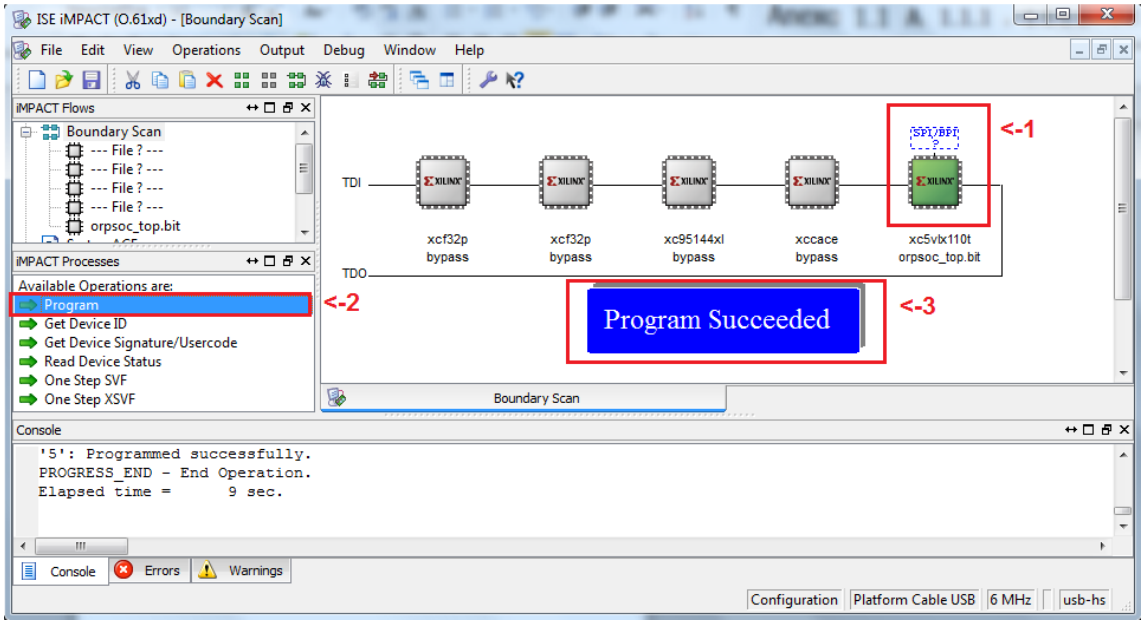


Figura 70: iMPACT – Programação da placa

Assim que terminar o projeto estará a executar na placa, caso se queira efetuar o 'reset' do processador, clicar no botão 'CPU Reset', Figura 71.

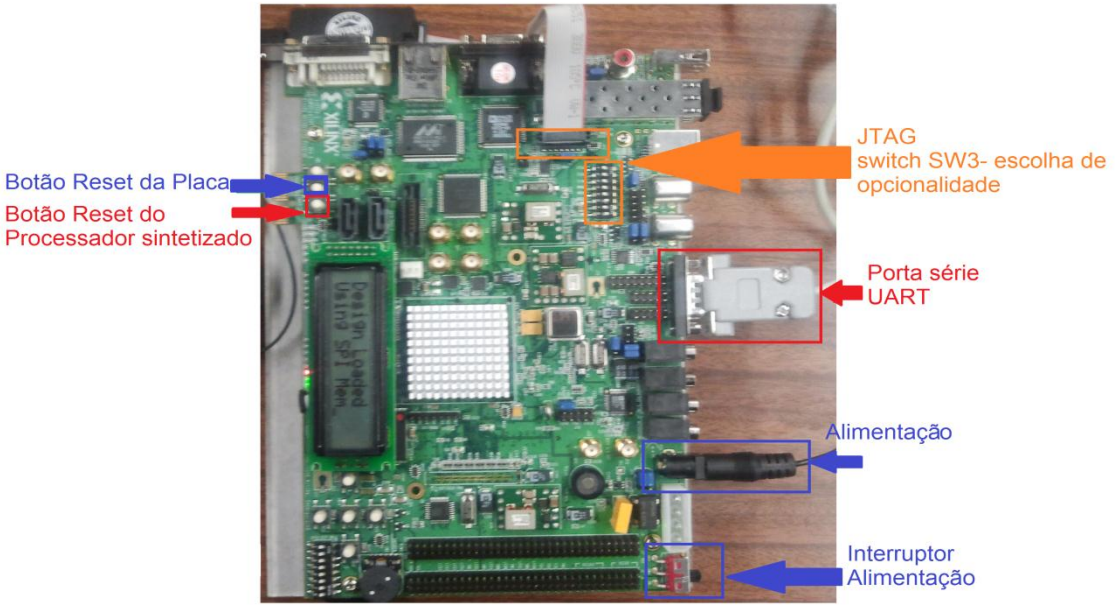


Figura 71: Esquema de ligação a placa XC5VLX110T